Program Comprehension Through Sonification

Lewis Irwin Berman

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

15 August, 2011

School of Engineering and Computing Sciences

Lewis Irwin Berman

Program Comprehension Through Sonification

ABSTRACT

Background: Comprehension of computer programs is daunting, thanks in part to clutter in the software developer's visual environment and the need for frequent visual context changes. Non-speech sound has been shown to be useful in understanding the behavior of a program as it is running.
Aims: This thesis explores whether using sound to help understand the static structure of programs is viable and advantageous.
Method: A novel concept for program sonification is introduced. Non-speech sounds indicate characteristics of and relationships among a Java program's classes, interfaces, and methods. A sound mapping is incorporated into a prototype tool consisting of an extension to the Eclipse integrated development environment communicating with the sound engine Csound. Developers examining source code can aurally explore entities outside of the visual context. A rich body of sound techniques provides expanded representational possibilities. Two studies were conducted. In the first, software professionals participated in exploratory sessions to informally validate the sound mapping concept. The second study was a human-subjects experiment to discover whether using the tool and sound mapping improve performance of software comprehension tasks. Twenty-four software professionals and students performed maintenance-oriented tasks on two Java programs with and without sound.
Results: Viability is strong for differentiation and characterization of software entities, less so for identification. The results show no overall advantage of using sound in terms of task duration at a 5% level of significance. The results do, however, suggest that sonification can be advantageous under certain conditions.
Conclusions: The use of sound in program comprehension shows sufficient promise for continued research. Limitations of the present research include restriction to particular types of comprehension tasks, a single sound mapping, a single programming language, and limited training time. Future work includes experiments and case studies employing a wider set of comprehension tasks, sound mappings in domains other than software, and adding navigational capability for use by the visually impaired.

Program Comprehension Through Sonification

Lewis Irwin Berman

Declaration

Unless clearly indicated otherwise, the work presented by this thesis is the author's. Parts of this work have been presented in academic conferences or workshops, and references to related publications are given below and at the appropriate sections. No part of this material has previously been submitted for a higher degree at Durham University or any other university.

The following publications were produced during the course of this thesis:

The Sound of Software: Using Sonification to Aid Comprehension. ICPC'06 [10]

Listening to Program Slices. ICAD'06 [11]

Integrating CSound with Eclipse: Listening to Software Under Development. CSound Journal, Winter, 2009 [12]

Using Sound to Understand Software Architecture. SIGDOC'09 [13]

Lewis Berman
15^{th} August 2011

Contents

Abstract **2**

Table of Contents **4**

List of Tables **9**

List of Figures **11**

Acknowledgements **13**

1 Introduction **14**

1.1 Introduction 14

1.2 Motivation 17

1.3 Propositions 18

1.3.1 Importance of the Propositions 19

1.4 Criteria for Success 22

1.5 Agenda 22

1.6 Summary 23

2 Program Comprehension **24**

2.1 Introduction 24

2.2 Software Maintenance 25

2.3 Program Comprehension Models 31

2.3.1 Top-Down Program Comprehension 32

2.3.2 Bottom-Up Program Comprehension 34

2.3.3 Opportunistic and Knowledge-Based Models 35
2.3.4 Integrated Model of Program Comprehension 37
2.3.5 Concept Assignment 39
2.3.6 Language Differences and Program Comprehension 39
2.4 Visual Tools for Program Comprehension 41
2.5 Summary 43

3 Sonification **44**
3.1 Introduction 44
3.2 Sonification and Auditory Display 45
3.3 Sonification in the Computing World 54
3.4 Spatial, Temporal, and Visual Metaphors 57
3.5 Design Guidelines 59
3.6 Listening, Processing, and Learning 63
3.7 Audio Engines and Tools 66
3.8 Sonification for Program Comprehension 68
3.9 Summary 76

4 Listening to Software Structure **77**
4.1 The Concept 78
4.1.1 Low-Level Java Structure 78
4.1.2 Sound Universe 81
4.1.3 Design Process 84
4.1.4 Reference Sound Realization 86
4.2 Feedback Results 96
4.2.1 Class Size Presentation 97
4.2.2 Entity Presentation 98
4.3 Summary 100

5 Prototype Tool Implementation **101**

5.1 Introduction . . . 101
5.2 The Tool . . . 104
5.3 Sound Construction . . . 107
5.4 Summary . . . 108

6 Review of Evaluation Techniques 109
6.1 Introduction . . . 109
6.2 Review of Possible Approaches . . . 112
6.2.1 Data Collection . . . 113
6.2.2 Coding, Analysis, and Experiment Design . . . 125
6.2.3 Evaluation Frameworks versus Empirical Techniques . . . 128
6.2.4 Chosen Approaches . . . 129
6.3 Summary . . . 129

7 Studies and Results 130
7.1 Introduction . . . 130
7.2 Methods . . . 130
7.2.1 Study One Method . . . 130
7.2.2 Study Two Method . . . 137
7.3 Results from Studies . . . 152
7.3.1 Study One Results . . . 152
7.3.2 Study Two Results . . . 163
7.4 Summary . . . 169

8 Analysis 173
8.1 Exploration and Task Performance . . . 173
8.1.1 CP Exploration, Unsonified . . . 174
8.1.2 CP Exploration, Sonified . . . 176
8.1.3 CP Task 5, Unsonified . . . 176
8.1.4 CP Task 5, Sonified . . . 177

8.1.5 Discussion of CP Exploration and Task 5 Strategies 178
8.1.6 PICT Exploration, Unsonified 179
8.1.7 PICT Exploration, Sonified 180
8.1.8 PICT Task 5, Unsonified 181
8.1.9 PICT Task 5, Sonified 181
8.1.10 Discussion of PICT Exploration and Task 5 Strategies . . . 183
8.1.11 CP Task 4, Unsonified and Sonified 184
8.1.12 Concrete versus abstract sounds 189
8.2 Threats to Validity 189
8.3 Other Considerations 192
8.3.1 PSSUQ Survey 192
8.3.2 Adoption Issues 194
8.4 Guidelines 195
8.5 Summary 196

9 Conclusions **197**
9.1 Introduction 197
9.1.1 Contribution 197
9.2 Criteria for Success 200
9.3 Propositions and Conclusions 202
9.4 Future Work 202
9.4.1 Tool 204
9.4.2 Mapping Sound 205
9.4.3 Advanced Ideas 206
9.5 Summary 207

Appendices **209**

A Study One Session Protocol **210**

B Study Two In-Session Training **213**

B.1 Description 213

B.2 Protocol 213

B.2.1 Set Up 213

B.2.2 Introduction 213

B.2.3 Packages 214

B.2.4 Classes and Interfaces 214

B.2.5 Methods 215

B.2.6 Extends, Instantiated-By, Referenced-By 216

B.2.7 Within vs. Outside Project Space 218

B.2.8 Practice 218

C Study Two Session Protocol **220**

C.1 Description 220

C.2 Researcher Instructions 220

C.2.1 Have on Hand 220

C.2.2 Then Do 221

C.2.3 Common Procedure for Both Programs 222

C.2.4 CP - Course Predictor 224

C.2.5 PICT - Pictionary Server Package 226

C.2.6 PSSUQ 228

Bibliography **229**

List of Tables

Table 2.1 Chapin's Classification of Maintenance Activities 28

Table 3.1 Box and line analogy, in increasing complexity 58

Table 3.2 Listening Modes, adapted from Tuuri 64

Table 4.1 Mapping bass drum to class size 83

Table 4.2 Mapping entities within or outside the project by audio distance 84

Table 4.3 Mappings of sounds to entity types 87

Table 4.4 Mappings of sounds to method and class characteristics 88

Table 6.1 Treatment groups 125

Table 7.1 Study One internal and external validity threats 136

Table 7.2 Population and task characteristics in the experimental scenario 138

Table 7.3 Treatment groups 139

Table 7.4 Minimum subject qualifications 140

Table 7.5 Program characteristics 142

Table 7.6 CP Tasks 145

Table 7.7 PICT Tasks 146

Table 7.8 Study Two internal and external validity threats 147

Table 7.9 Subject profile 153

Table 7.10 Mapping features and their degrees of success 159

Table 7.11 Subjects, ordered by group 164

Table 7.12 CP Task 5 times, by group 165

Table 7.13 PICT Task 5 times, by group 165

Table 7.14 Kolmogorov-Smirnov tests for normality 166
Table 7.15 PICT Task 5 possible outliers 166
Table 7.16 Fisher results for CP Task 5 correctness 167
Table 7.17 Fisher results for PICT Task 5 correctness 167
Table 7.18 Kruskal-Wallis results 168
Table 7.19 PSSUQ Results 171

Table 8.1 KLM analysis of CP Task 4, unsonified 186
Table 8.2 KLM analysis of CP Task 4, sonified 187
Table 8.3 KLM estimates vs. actual task durations 187
Table 8.4 Threats to validity 192
Table 8.5 PSSUQ summary statistics 193

Table 9.1 Propositions and Conclusions 203

List of Figures

Figure 1.1 The Eclipse integrated development environment 15

Figure 2.1 Maintenance Types Decision Tree 29

Figure 3.1 Earcons within a family 47

Figure 4.1 Metaphor for software entities 82

Figure 4.2 Design process 85

Figure 4.3 Two writers as strictly upward patterns 91

Figure 4.4 Close vs. distant entities 93

Figure 4.5 Pitch mappings to size 97

Figure 4.6 Drum mappings to size in number of methods 98

Figure 4.7 Early concept 99

Figure 5.1 Eclipse user interface for sonification 102

Figure 5.2 System Architecture 105

Figure 7.1 Java search feature of Eclipse 143

Figure 7.2 Experiment flow, from subject's perspective 144

Figure 7.3 CP Task 5 and PICT Task 5 Box Plots 165

Figure 7.4 CP Task 4 box plots 168

Figure 7.5 Box plots for PSSUQ questions 170

Figure 8.1 Exploration strategies 174

Figure 8.2 Partial CP Class Diagram 175

Figure 8.3 Partial PICT Class Diagram 179

Figure 8.4 Initial state of environment for CP Task 4 185

ACKNOWLEDGEMENTS

I would like to thank:

Professors Malcolm Munro and Keith Gallagher, my supervisors, for providing me with the opportunity to undertake a Ph.D. and then supporting me throughout.

The members of my committee, for providing and enforcing a standard of quality.

Professors Roger Eastman, David Binkley, and the faculty and staff of the Computer Science Department of Loyola University Maryland, for their encouragement, help with LaTex, proofreading, and other assistance.

Professor Christopher Morrell, for statistical assistance and review.

Michael and LeDonna Berman, for their encouragement and support.

Marion Ehrlich, Robert Goetz, and Mary Jean Goetz, also for their encouragement and support.

This thesis was generated using LaTeX 2_ε. Examples in musical notation were prepared with the *MusixTex* set of macros.

Chapter 1

Introduction

1.1 Introduction

Program comprehension can be a daunting process which involves many tasks and decisions and consumes much information. The information is expressed visually, usually via text. An integrated development environment (IDE) such as Eclipse [45] tends to become cluttered as increasing numbers of windows are opened and more information is displayed. Multi-modal representation, including the introduction of sound as an additional information channel, has been shown to alleviate problems induced by clutter, though a mature methodology for developing multi-modal interfaces has not yet been achieved [117]. This thesis explores whether non-speech sound to represent software constructs can be of assistance by supplementing visual information for program comprehension.

Figure 1.1 shows the Eclipse IDE in typical use for maintaining a Java [59] program. In Figure 1.1, the developer is working on an expense recording program using a Java view, one of several commonly-used Eclipse layouts. The persistently-displayed Package Explorer[1] on the left provides navigation and a frame of reference within the *Expenses* project. It mixes a common folder/file metaphor with

[1]The Package Explorer is nearly identical to another class browser known as the Project Explorer. They are equivalent as employed herein.

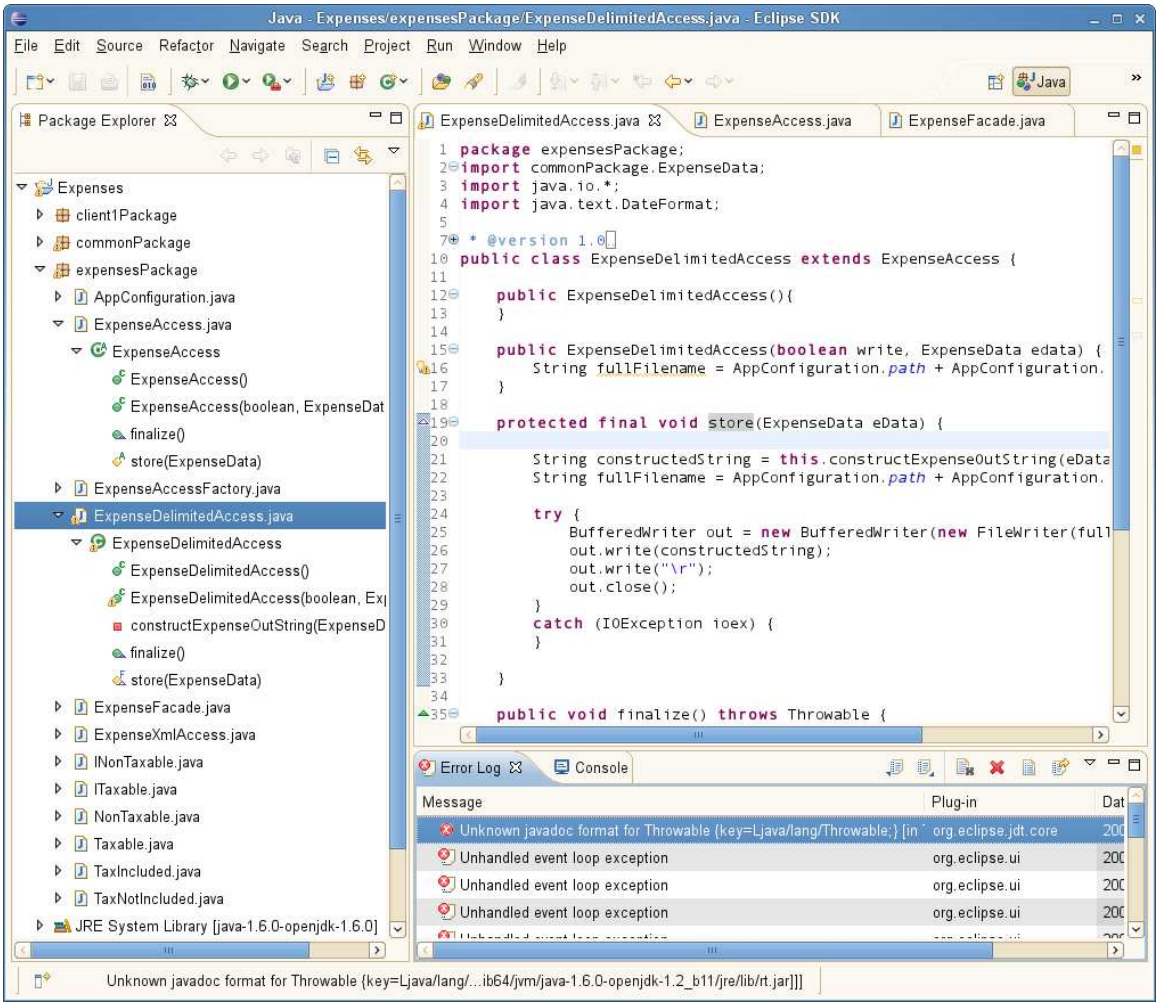

Figure 1.1: The Eclipse integrated development environment

Java constructs such as packages and classes. A file containing the *ExpenseDelimitedAccess* class has been selected by the developer and is displayed in the editor window to the right. Tabs indicate that files containing other classes, *ExpenseAccess* and *ExpenseFacade*, are being edited but are hidden behind the current window. The editor window competes for screen real estate with the Project Explorer and a window containing several tabs at the bottom, one tab showing errors in the project and the other behind it for run-time console messages. It is possible for a developer to have a dozen or more open editor tabs and four or more tabs in the bottom right area.

The Package Explorer shows the parent-child structure of entities within the project, from packages such as *expensesPackage* down through methods and class variables. In typical use, packages and classes are displayed and lower-level information is hidden until expanded on demand. Expansion lowers the number of classes that are visible in the Package Explorer. The *ExpenseAccess* and *ExpenseDelimitedAccess* classes have been expanded by the developer, revealing that each has overloaded constructors, *finalize* and *store* methods, and for the latter class, an additional method called *constructExpenseOutString*. The arguments of each method can be seen with a bit of horizontal scrolling. One must visit the *ExpenseAccess.java* editor tab to see that it is an *abstract* class, and one must visit the *ExpenseDelimitedAccess.java* editor tab to see that the abstract class is therein implemented. This tab also reveals that a *BufferedWriter* is instantiated and that its child methods such as *out.close* are called. A mouse hover would reveal text indicating that these belong to the *java.io* package external to the project. *Java.io* and its methods are aptly named to tell the developer they write data and close a data stream; this is not always the case for referenced entities.

Quite a bit of information is present in the IDE, and the developer must change visual focus, scroll, click hidden tabs, or navigate away from the primary visual context to access much of it. If sound can be shown to help describe characteristics

of a software entity of interest or identify and characterize related entities external to the current visual context, it may hold the promise of streamlining the program comprehension process by exploiting the potential of the human aural channel.

To explore that potential, a tool has been developed which exposes a mapping from static software structures to sound patterns. The sounds contained within the patterns describe structural and functional aspects of Java classes, methods, interfaces, and packages. Two studies have been performed using the sound mapping and the tool. The first study was exploratory, focusing on recognition and interpretation upon hearing the sound patterns. The second study was a quantitative experiment, focusing on two specific program comprehension tasks.

1.2 Motivation

It may be possible to improve program comprehension by adding sonification in hitherto unexplored ways. *Sonification* is defined as "the use of non-speech audio to convey information."[87] It is a subset of auditory display [69], the use of sound, generally non-speech, for understanding, alerting, warning, and real-time control. Research in *Program comprehension* studies the acts and mental models involved in understanding a new or existing computer program [175]. The two research areas have met mainly in studies of the comprehension of dynamic aspects of a program, that is, the behavior of a program when it runs, less so in comprehension of static structure, the fixed construction of a program. Chapter 2 summarizes applicable program comprehension research. Chapter 3 summarizes applicable sonification research.

This thesis addresses whether sound can supplement the visual mode of comprehension of the static structure of a program by reinforcing information or providing it in an orthogonal or more convenient manner.

1.3 Propositions

The overall research question below captures the problem at hand.

> Can sonification supplement visual information to support comprehension of the low-level, static structure of non-trivial computer programs?

"Support" has two implications:

1. that program comprehension can be performed by substituting sonification for some visual means, and
2. that using sonification presents an advantage in program comprehension over purely visual means.

Subordinate questions, which shall be referred to as propositions,[2] are

1. A consistent, comprehensible mapping of non-speech sound patterns to the static entities of a software system can be devised.
2. The mapping can be used to identify software entities.
3. The mapping can be used to characterize software entities and their relationships when encountered.
4. The mapping, incorporated into an integrated software development environment, can be used in the performance of program comprehension tasks.
5. Using the mapping in a multimodal software development environment can improve performance of software comprehension tasks over that using a software development environment without sound.

[2]Not to be confused with propositions in formal logic, these are postulations whose truth supports an affirmative answer to the research question, after Yin [183].

Propositions 1, 2, and 3 concern the sound mapping. Their validity or lack thereof is suggested by the first study and reinforced by the second. Propositions 4 and 5, concern the use of the tool to facilitate program comprehension tasks. They are tested through the second, experimental study.

1.3.1 Importance of the Propositions

The impact on program comprehension and sonification is considered.

- *Program Comprehension.* The last proposition implies that program comprehension may be accomplished faster when the auditory sense is involved than by visual means alone. The propositions taken together offer the possibility that sonification may lead to more efficient or effective program comprehension in general. As understanding complete and partially complete programs is part of the software maintenance process, positive results may ultimately decrease the cost of developing or maintaining a computer program and improve the resulting program's quality.

- *Auditory Display.* Proposition 1 implies that it is possible to represent relatively abstract items, in this case software entities and their characteristics, using a coherent system of sounds and sound combinations. This stands in contrast to representation of concrete items and concepts (such as an automobile or a war) by intuitively associated sounds (such as a motor or a bank of cannon, respectively). The concept of an entity being a software class or method, and a method furthermore belonging to its class or instances of its class, has no such intuitive audio analog. The first three propositions state that an audio representation for entities or concepts at the higher level of abstraction can be usable. Comprehension of analogous areas outside the software domain may benefit. More immediately, development of a tool supporting a sound mapping, will contribute to the understanding of sound mapping design and sound mapping design patterns.

The following subsections restate and discuss the five propositions.

Proposition 1

> A consistent, comprehensible mapping of non-speech sound patterns to static components of a software system can be devised.

Sounds, notably auditory icons [53], can represent real-world entities directly. The most intuitive entities feature audio, like an alarm clock or an old-style telephone bell. Almost as intuitive are things having concrete audio associations, such as a battle (with cannon shots) or a seashore (with crashing surf). Sonified line graphs [28], though not auditory icons, are still intuitive, possibly after a bit of explanation, as pitch is directly and continuously mapped to the Y value as a function of X. In the problem at hand, some concrete notions can be used, for example, a door closing for a "close" method. However, notions such as class and method have neither concrete connotations nor continuous or discrete data values. Characteristics such as *static* and *accessor* have weak aural connotations at best.

Proposition 2

> The mapping can be used to identify software entities when they are aurally encountered.

The following question arises:

- To what extent can identification of software entities be understood?

Proposition 3

> The mapping can be used the characterized software entities and their relationships when encountered.

The following question arises:

- To what extent can characterization of software entities be understood?

Proposition 4

> Such a mapping, incorporated into an integrated software development environment, can be used in the performance of program comprehension tasks.

The following questions arise:

- Can triggering the sounds, hearing them, and performing the necessary cognitive processing be performed in a timely and usable manner?
- Can the addition of sound be of low enough cognitive cost to render it desirable?
- Can it be made usable in both exploratory and more targeted modes of operation?
- Can a sound mapping be useful in program comprehension given limited training, especially when the required variation of sounds exceeds the number of available aural metaphors?

Proposition 5

> The mapping can improve performance of software comprehension tasks over that using a software development environment without sound.

- Does this apply across types of programs, roles, and comprehension tasks?
- Does musical ability have an impact on performance?

The ultimate goal is to improve the state of the art in program comprehension by exploiting sonification. The solution herein is motivated by a focus on low-level, static program structure: the existence of, characteristics of, and relationships among packages, classes, interfaces, and methods in a Java environment.

Improvement can occur over several dimensions: speed of completing comprehension tasks, accuracy, and depth and breadth of understanding. Speed is tangible and easily measured within an experimental context. Accuracy necessarily has a relationship to task duration, so an evaluation of accuracy can be made along with speed when applied to tasks resulting in finite answers to specific comprehension questions. To measure depth of understanding would require larger programs as experimental objects, extended and possibly multiple coordinated tasks, and a large and possibly subjective data collection instrument. Breadth of understanding and retention would have similar requirements and be similarly less tangible than speed. Therefore, speed is chosen as an initial measure of interest.

1.4 Criteria for Success

This research has succeeded when the following criteria are met. Each criterion maps to the proposition in parenthesis.

1. Define a reference sound mapping to static software entities for one programming language. (Proposition 1)
2. Evaluate the reference sound mapping concept. (Propositions 2, 3, and 4)
3. Apply the mapping to program comprehension tasks. (Propositions 4 and 5)
4. Develop a prototype demonstrator tool using an instance of the reference sound mappings. (Proposition 4)

1.5 Agenda

This thesis is organized into nine chapters.

Chapter 1, Introduction briefly introduces the problem, propositions concerning the problem, and the structure of this thesis.

Chapter 2, Program Comprehension is a summary of software maintenance types and a review of the literature on program comprehension models.

Chapter 3, Sonification is a review of the literature on audio display and sonification, especially as applied in software engineering.

Chapter 4, Listening to Software Structure describes the solution concept.

Chapter 5, Prototype Tool Implementation describes the prototype tool developed and used experimentally.

Chapter 6, Review of Evaluation Techniques is a review of the literature on human-subjects evaluation in software engineering and notably in program comprehension.

Chapter 7, Studies and Results sets forth two studies and reports their results. Each study and its protocol is described, then the results of each are reported in turn.

Chapter 8, Analysis provides analysis and discussion of the results of the study.

Chapter 9, Conclusions provides conclusions and suggests future research directions.

1.6 Summary

This chapter has described the problem of visual information overload in a software maintenance environment. It has proposed the use of sound during static display of source code to alleviate visual information overload. A research question and a set of five propositions has been articulated. It has laid out a set of propositions concerning the problem and solution, laying the groundwork for the remainder of the thesis, which investigates the sonification of software.

Chapter 2

Program Comprehension

This chapter presents the program comprehension literature and related topics of significance concerning software architecture, software maintenance, and software visualization. Section 2.1, Introduction, introduces these areas and their relationship to the problem at hand. Section 2.2, Software Maintenance, introduces software maintenance and enhancement. Section 2.3, Program Comprehension Models, introduces models of human program comprehension. Section 2.4, Visual Tools for Program Comprehension, discusses tools, notably those that support visualization and how visualization relates to sonification. Section 2.5, Summary, concludes the chapter.

2.1 Introduction

Program comprehension is essential for successful software maintenance. Similar systems and reusable components may have to be understood as early as initial development of a software system.

In this thesis, a person performing program comprehension is referred to as the *developer* or *maintainer*, as further software development and maintenance are the usual reasons for undertaking program comprehension. It is understood that persons other than developers may perform program comprehension tasks for quality

assurance, auditing, estimating, and other purposes. Software designers must understand the software architecture of the system to effect modifications that are correct and maintainable. Maintainers must understand control-flow and data-flow interdependencies among the software units. Understanding extends from the low-level code through data and control structures and finally the application domain.

2.2 Software Maintenance

"Maintenance and enhancement are generally defined as activities which keep systems operational and meet user needs [95]." Physical systems require maintenance, in general, to negate the effects of aging and wear, returning them to their original state. Software suffers no such physical effects. Rather, it must be adapted to changes in its operational domain, including explicit and implicit user expectations [88]. Software maintenance is currently defined by the International Standards Organization (ISO) as "the totality of activities required to provide cost-effective support to a software system [71]." A somewhat less succinct definition, relevant to software, is that encountered for system maintenance:

> the modification or upkeep of information system hardware and software to sustain or improve performance, to correct faults or deficiencies, or to adapt the system to changes in environment or requirements. [159]

When user needs result in new functional requirements, enhancements are performed. The phrases *program evolution* and *software evolution* are employed to encompass the progression of a software system through maintenance, including enhancement [89]. In practice, "maintenance" is often used interchangeably with "software evolution."

Swanson characterized three types of maintenance activities according to the purpose of the maintainer [162]. The types are corrective maintenance, adaptive maintenance, and perfective maintenance. Corrective maintenance is performed in response to processing failures, such as the abnormal termination of a program, performance failures, the inability to meet performance criteria, and implementation failures, such as not meeting the functional specification. Adaptive maintenance is performed so that the software product can run correctly in a new processing environment or in response to changes in data formats from other systems and media. Perfective maintenance is performed to "make the program a more perfect design implementation" by eliminating processing inefficiencies, enhancing performance, and improving maintainability. Lientz and Swanson added user enhancements and improved documentation to perfective maintenance activities, invoking the combined phrase *maintenance and enhancement* [94][95].

Lientz and Swanson discovered via industry surveys that perfective maintenance, including enhancements for users, accounted for over half of the total maintenance effort [94]. They also found that less than half of the individuals assigned to maintenance of an application had worked on the initial development of the system, that approximately three-fourths of applications were maintained by one full-time person or less, that at least half of the applications were maintained by a half-time equivalent or less, and that maintainers of an application often had other assignments as well. Two implications of these findings are that application knowledge must be communicated from developers to maintainers, and that maintainers may have to refresh their own knowledge after periods of inactivity with a particular application. Lientz and Swanson confirm that the maintenance they measure "consists largely of continued development..." [p. 151]

A fourth type of maintenance, preventive maintenance, indicates maintenance performed to avoid potential, possibly unanticipated failures [71]. The International Standards Organization (ISO) recognizes *improvement* as a type of main-

tenance, replacing the term *perfective* [70].

Von Mayrhauser and Vans listed five types of tasks performed during software evolution [175]. The types are categorized by the purpose of the maintainer or developer:

1. Adaptive maintenance - adapt a solution to meet new requirements.
2. Perfective maintenance - improve a solution to better meet its requirements.
3. Corrective maintenance - correct errors in a solution.
4. Reuse - identify and integrate reusable components into a solution.
5. Code leverage - reconfigure an existing set of reusable components into a new solution.

Each task type is characterized by a set of activities. *Understanding* is the first activity listed for all task types. Maintenance tasks require the developer to understand the system. Reuse and Code Leverage tasks require the developer to understand the problem as well as at least some portion of an existing code base. Ideally, a developer evaluating a reusable code library will only have to understand the class and method specifications which comprise its application programming interface (API).

Chapin et. al. proposed "a finer grained objective classification of the types of activities involved in software evolution and software maintenance [32]." Chapin's typology is grounded in observation and artifacts rather than in goals as expressed by maintainers. Mutually-exclusive types are grouped into clusters. An activity may be an aggregate of types, in which one type can be considered to be dominant. The clusters and types, in rough order by impact on the software itself, are shown in Table 2.1.

A decision tree is traversed to determine which type(s) of maintenance have been performed. Figure 2.1, depicting the decision tree, is an adaptation of a diagram originally by Chapin et. al. [p. 10, fig. 2]. Swanson's corrective and adaptive

A. Support Interface Cluster
   1. Training
   2. Consultive
   3. Evaluative

B. Documentation Cluster
   1. Reformative
   2. Updative

C. Software Properties Cluster
   1. Groomative
   2. Preventive
   3. Performance
   4. Adaptive

D. Business Rules Cluster
   1. Reductive
   2. Corrective
   3. Enhancive

Table 2.1: Chapin's Classification of Maintenance Activities

maintenance categories remain intact. Perfective maintenance becomes reformative, updative, groomative, preventive, performance, reductive, or enhancive.

Consider that some logging code has been modified to combine multiple logging classes into one. No functionality has been added, removed, or altered. The decision tree is traversed in the following sequence:

1. *A: Was software changed?* Yes.
2. *B: Was source code changed?* Yes.
3. *C: Was function changed?* No.
4. *C-1: Did the code change make the software more maintainable?* Yes, therefore the maintenance type is Groomative.

The ISO's software maintenance standard defines the requirements for an organization's maintenance process, which extends to problem reporting, verification, and management activities [71]. It lists six major maintenance activities, the second through fourth of which form the core lifecycle of a problem or modification:

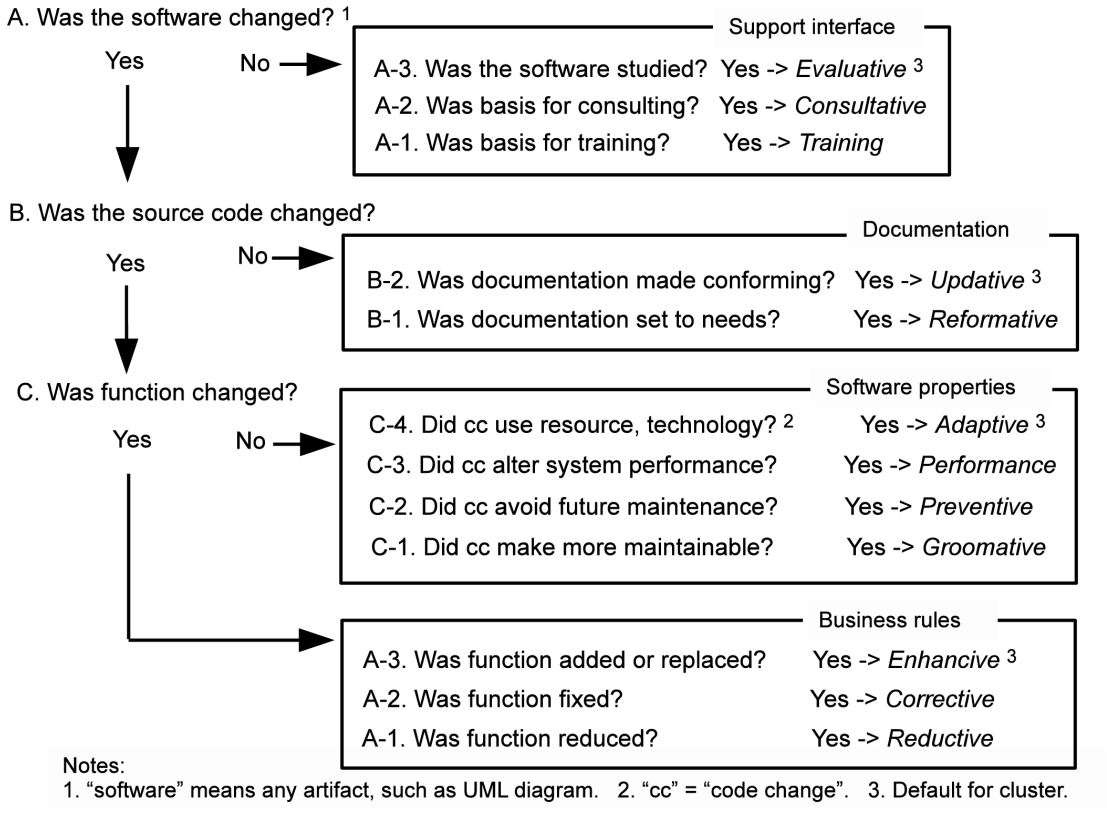

Figure 2.1: Maintenance Types Decision Tree

1. *Process Implementation* - define the maintenance process for the software project.

2. *Problem and Modification Analysis* - develop options for implementing the modification. Determine if the modification is a correction or enhancement. If a correction, determine if it is corrective or preventive. Enhancements may be adaptive or perfective.

3. *Modification Implementation* - perform and verify the modification.

4. *Maintenance Review/Acceptance* - ensure that the modifications are correct and developed according to standards.

5. *Migration* - Adapt the system to a new environment, which may require its own software development lifecycle.

6. *Software Retirement* - Plan and execute retirement of the system.

The idea of "evolution" has become strengthened as delivery has become less of a one-time project milestone. Under the waterfall model of software development, software was specified and designed prior to being implemented (coded), tested, and released [42]. A maintenance lifecycle phase commenced after release to the customer. More recently, spiral and other iterative lifecycle models have caused maintenance phases to overlap with new development [20]. Most recently, agile methods have emphasized short development iterations, each culminating in product delivery, and intentional development of software that will undergo future refactoring [65]. These advances have blurred the distinction between development and maintenance. The first delivery of an application may perform three or four functions and have a simple, one-frame user interface. As more user interface elements are added for further deliveries, existing elements may undergo perfective maintenance in their naming and mutual structuring to accommodate a larger set of elements. Already-functioning modules of code may be altered as it is noticed that they should fit into a standard design scheme. Maintenance is occurring even while development is continuing, and it may be difficult to describe an activity as belonging to one or the other.

"Program understanding has long been recognized as a central activity in a variety of maintenance tasks [177]." Reverse engineering and ripple analysis are two activities in which program comprehension is clearly required [57]. Reverse engineering is undertaken when systems suffer from inadequate or out of date documentation, or to verify the existing documentation. Maintainers perform ripple analysis to determining the effects that a modification will have upon other parts of the software system. To describe and effect modifications, modules (e.g., objects in an object-oriented system) must be selected, and at a lower level the modules must be understood.

Various program comprehension models have been proposed and tested to describe program understanding. They are explored in the following section.

2.3 Program Comprehension Models

Narrowly, *comprehension* is one level of Bloom's Taxonomy of Education Objectives, as articulated by Kelly and Buckley:

> The comprehension category represents the lowest level of understanding. This is where someone understands what is being represented and can translate the representation into different terms. For example, if a programmer can summarize a section of code then he is demonstrating comprehension. [77]

Program comprehension, according to Kelly and Buckley, is commonly used to refer more broadly to *understanding*, collecting *knowledge*, applying *synthesis* to previously learned material, and performing *analysis*. Vinz implies the incremental nature of program comprehension:

> Program comprehension involves the process of extracting properties from a program in order to achieve a better understanding of the software system. [172]

The information sources that serve as input to the program comprehension process may be the source code of interest, code from other systems for comparison, textual descriptions of the design, graphic representations of the design, metrics derived from the code, and similar items of interest. The source code itself consists of executable and non-executable program statements, comments, identifiers within source statements, and similar constructs. The information sources may be examined either statically or dynamically. Static examination is accomplished by reading The source code and other sources to discover the program's fixed structure as described above. Dynamic examination is accomplished by running the program (either in actuality or on paper/in thought) to discover the program's state at given points in its execution. In something of a hybrid approach, ex-

ecution of part of the program is simulated by the maintainer while it is being examined statically.

A maintainer forms a *mental model* [175] of a program as its code and documentation are assimilated. The mental model is built using cognitive processes, existing knowledge, and other information structures that together form a *cognitive model*. The following subsections describe the major cognitive models in the program comprehension literature.

2.3.1 Top-Down Program Comprehension

Brooks proposed a top-down theory of program comprehension based on formulation and validation of successive hypotheses [27]. The maintainer reconstructs domain knowledge and maps that knowledge to the code. The comprehension process is initiated with an often-sketchy primary hypothesis specifying probable program inputs, program outputs, major data structures, and major processing sequences. The maintainer forms subsidiary hypotheses in a depth-first manner, accepting or rejecting them as the code is examined. New information obtained in this manner can be used to form subsidiary hypotheses elsewhere in the hypothesis space. Brooks posits that the maintainer's hypothesis-solving technique is top-down in order to reduce cognitive load.

According to Brooks, hypothesis verification is aided by *beacons* [27] in the code. Beacons are "sets of features that typically indicate the occurrence of certain structures or operations." [27] Different beacons may have stronger or weaker probabilities of indicating a structure or operation. As beacons are uncovered, not only are existing hypotheses verified, rejected, or modified, but new hypotheses are formed from the broader knowledge gained.

Wiedenbeck demonstrated experimentally that beacons play a large role in the initial stages of program comprehension [179]. Beacons were better recognized by those with more programming experience than by novices, consistent with the

idea that they have learned a larger store of idiomatic patterns. Incorrectly-placed beacons hampered understanding by experienced programmers, consistent with the idea that experienced programmers may initially find beacons and tentatively confirm hypotheses without reading the surrounding code in detail. Wiedenbeck's experiment was based on recall tasks and recognition using code fragments (e.g., a seventeen-line sort procedure), not on a situation involving understanding a large or complete program.

Soloway and Ehrlich observed top-down program comprehension in two empirical studies when the code is familiar [149]. Subjects matched code they examined to *programming plans*, stereotype code fragments representing known action sequences. Subjects also employed *rules of programming discourse*, rules based on expectations of conventions in the code. Experienced programmers performed better than novices unless rules of programming discourse were violated in constructing the plans, in which case performance was similar. Soloway and Ehrlich suggest, based upon their observations, that the mental model of the program involves forming a schema-based hierarchy goals and programming plans. Like Wiedenbeck, Soloway and Ehrlich did not observe a "real world" program comprehension situation, instead basing their first experiment on fill-in-the-blank tasks and the second on total recall tasks.

Koenemann and Robertson conducted an experiment that affirmed program comprehension as primarily a goal-oriented, hypothesis-driven process [85]. They noted the use of beacons, especially in the form of procedure and variable names. Their experiment involved four maintenance tasks performed among twelve experienced programmers. Results indicated that maintainers study only that code deemed relevant on an as-needed basis, as determined via an opportunistic strategy. Evidence was found of three degrees of relevance:

1. *Directly* relevant code is that which must be modified or copied and edited. This code is studied in the greatest detail.

2. *Intermediately* relevant code includes code segments perceived to interact with the directly relevant code. This code is studied in less detail than directly relevant code.

3. *Strategically* relevant code guides the comprehension process by enabling the maintainer to locate the other relevant parts of the code. It is not studied in detail.

The experimental subjects had been provided with program documentation, including high-level program descriptions. This may have influenced the choice of a top-down strategy.

2.3.2 Bottom-Up Program Comprehension

Bottom-up models of program comprehension rely on the concept of *chunking* information into groups to be able to store more information in short-term memory [110]. In bottom-up approaches, such as that of Pennington [121], maintainers read the code, then group the code statements to form higher-level abstractions. The abstractions themselves are chunked into abstractions at even higher levels, until the program as a whole is understood. Pennington identifies a *program model* and a *situation model*. The program model is built first, as the maintainer forms chunks [110], basic units of retainable information, from syntactic structures and cross-references the chunks. The program model contains the maintainer's idea of sequential control flow within the program. As the program model matures, the situation model is built, applying domain knowledge to the bottom-up control-flow abstractions to result in a model containing data-flow and functional abstractions. Pennington indicates that program model constructs exist at the *microstructure* level, consisting of text structure knowledge, while situation model constructs exist at the *macrostructure* level where plan knowledge is operative.

Pennington's subjects were professional programmers, having reasonably well developed pre-existing text structure and plan knowledge. Pennington noted that

the comprehension task likely influences the comprehension strategy. In Pennington's experiment, maintainers were first asked to read code for general understanding, then embark on a maintenance task. It was only during the maintenance task that maintainers developed the situation model. Pennington also notes that her experiments involved only small programs. Pennington treats programs as text (the text being the source code), which precludes introduction of documentation as an understanding aid. She suggests that the presence of documentation concerning the application domain might promote earlier construction of the situation model.

2.3.3 Opportunistic and Knowledge-Based Models

Littman, Pinto, Letovsky, and Soloway determined that maintainers use either a *systematic* or *as-needed* comprehension strategy [96]. A systematic strategy is used to understand program behavior prior to attempting a modification. Through a systematic strategy, the maintainer explores data flow and control flow paths among different procedures. An as-needed strategy is employed to minimize time studying the program. It is localized to the extent that it is "unlikely to detect interactions in the program that might affect or be affected by the modification." A distinction is made between two forms of knowledge:

1. *static knowledge*, the maintainer's knowledge of the program's actions, functional components, and the objects upon which it operates, and

2. *causal knowledge*, the maintainer's knowledge of how structurally separate parts of the program interact.

Experimental results indicated that the error rate resulting from code changes was greater for maintainers who employed only an as-needed strategy due to the failure to develop sufficient causal knowledge.

Letovsky concluded through empirical research that humans performing program comprehension use top-down and bottom-up strategies upon discovering cues appropriate to one or another [92]. Letovsky's model has three components:

1. *knowledge base* - the maintainer's existing knowledge of the application domain, programming domain, programming plans, program goals, and rules of discourse.

2. *mental model* - the maintainer's understanding of the program during comprehension.

3. *assimilation process* - the process by which the maintainer's mental model evolves, incorporating the maintainer's knowledge base along with the program's source code and documentation. The process may be top-down, bottom-up, or a combination of both.

Letovsky identifies the activities which inform the assimilation process as *inquiries*. The maintainer actively or tacitly asks a question, poses a conjecture, and performs search to verify or reject the answer. Questions are categorized into five rough types:

1. *why* - what is the purpose of this design?

2. *how* - how is a program goal or subgoal achieved?

3. *what* - what does a function, variable, or construct do?

4. *whether* - does the program behave one way or another?

5. *discrepancy* - note an apparent inconsistency in the source code.

Conjectures are plausible inferences that attempt to answer an explicit *why*, *how*, or *what* question. A subset of *what* conjectures consists of *word* conjectures which pertaining to the meaning of identifiers in the program.

2.3.4 Integrated Model of Program Comprehension

Von Mayrhauser and Vans extracted the common elements of preceding theories of program comprehension, specifically involving code cognition, to yield a meta-theory [175]. Their model covers the kinds of knowledge involved in code cognition, the mental model that is refined during code cognition, strategies aimed at refining the mental model, and inferences, hypotheses, and tests that validate or modify the mental model.

General knowledge and knowledge specific to the application under investigation are needed in the code cognition process. General knowledge covers such areas as the programming language, common algorithms, and possible approaches to the solution at hand. If an integrated development environment (IDE) is used, as is commonly the case for sizable programs, the developer's general knowledge includes editors, browsers, and other tools. Such general knowledge would also include how to assimilate and interpret the information presented by any visualization or sonification used by the developer.

Specific knowledge is built over the code cognition process, during which the programmer builds and refines a mental model of the program. The mental model is built top-down through the refinement of plans, beginning with the top-level plan of the program, and bottom-up through the successive understanding of chunks, the bottom-most being localized text structures. They are the schemas in a schema-slot concept of knowledge representation. Formulation and testing of hypotheses either validate plans in the mental model or cause them to have to be revised. To test a hypothesis about a plan, the developer formulates a strategy for understanding and dealing with chunks of program information. The developer also cross-references knowledge at different levels of abstraction.

Suppose the developer is presented with the project shown in Figure 1.1. Seeing a class named *ExpenseDelimitedAccess*, the developer may hypothesize from his or her knowledge base that this class constructs a string of delimited values

and writes the string to a file using standard I/O routines. A look at its methods confirms that it constructs a delimited string, as one of the methods is called *constructExpenseOutString*. A look at the *store* method in the editor pane confirms that the output occurs via the standard *java.io* package by constructing a *BufferedWriter* object and calling its *write* and *close* methods. Verifying the hypothesis has involved expansion of the class in the browser, bringing up the class in the editor pane, looking at the class' imports, examining the *constructExpenseOutString* method, examining the *store* method, and cross-checking that method with the imported packages.

An expert developer will have at his or her disposal a mental repository of plans as pre-existing templates. For example, the expert is likely to know about a variety of design patterns involving multiple objects and their relationships. The expert can match an observed program chunk to a given design pattern, even if there is a variation from the template. The expert can develop a mental model using breadth-first strategies, while the novice will operate in a mostly bottom-up manner, starting with localized control flow.

The integrated code comprehension model of von Mayrhauser and Vans has four components: a knowledge base, a top-down model, a situation model, and a program model. The knowledge base holds the developer's actual pre-existing and newly assimilated knowledge, and it informs the other models. the top-down model reflects top-down refinement of plans. The program model reflects bottom-up chunking. The situation model relates the program to its problem domain. Beacons are useful in formulating all three comprehension models.

Von Mayrhaurser and Vans confirmed through observation in an industrial environment that, for large programs (in one case, 90,000 lines of code), developers who will be maintaining the code prefer to begin program understanding at a relatively high level of abstraction and proceed top-down, switching between components to understand the architecture [176]. As a developer continues to lower

levels, bottom-up strategies come into play.

2.3.5 Concept Assignment

Concept assignment is a key idea in program comprehension. Biggerstaff describes concept assignment as

> a process of recognizing concepts within a computer program and building up an "understanding" of the program by relating the recognized concepts to portions of the program, its operational context and to one another. [14]

Biggerstaff differentiates between *programming-oriented concepts* such as sorts and structure transformations and *human-oriented concepts* such as airplane seat reservations. While reading code, the former are easier to recognize. The latter are often delocalized within the code.

Rajlich observes that "the knowledge of domain concepts is based on program use [128]." If a domain (human) concept is a program feature, that is selectable by the user, a technique for mapping it to code is to run the program twice, once with and once without invoking the feature, and note which code only runs only when the feature is selected. This is known as *software reconnaissance* or *dynamic search*. A static method of locating a feature is to search through the code, following control-flow and data-flow dependencies. This may consume significant time and effort.

2.3.6 Language Differences and Program Comprehension

Pennington observed that programming language biases the program comprehension process [121]. Maintainers more familiar with COBOL performed better answering data-flow questions, while those more familiar with FORTRAN performed better answering control-flow questions. Bergantz and Hassell affirmed

construction of both a program model and a domain model during comprehension of PROLOG programs [8].

Objects in an object-oriented language often encapsulate domain concepts or design-specific concepts. Classes are often named for the domain concepts they represent. Programming plans are delocalized, passing through multiple classes. Corritore and Wiedenbeck observed differences between maintainers performing comprehension of equivalent programs written in procedural and object-oriented languages [37]. Using Pennington's model as a framework, they investigated early and late stage comprehension of C programs by experienced procedural programmers and C++ programs by experienced object-oriented programmers. After a period of code reading, the procedural programmers had developed an incomplete but "balanced" model of the C code, while the OO programmers had developed a similarly incomplete but highly domain-weighted model of the C++ code, the latter performing notably well on structure. After an ensuing maintenance task, both procedural and OO programmers exhibited a balanced model. Results also indicated a difference among procedural programmers from Pennington's observations in initially developing a balanced rather than program-weighted model. Corritore and Wiedenbeck postulate that larger program size (approximately 800 lines of code, four times higher that of the small programs in Pennington's study) motivated the difference.

Shaft [141] observed that some maintainers use *metacognition* during program comprehension. That is, they deliberately choose a comprehension strategy for a given subtask, then monitor their progress on the subtask. General use of metacognition appears to reduce comprehension when working in an unfamiliar domain.

2.4 Visual Tools for Program Comprehension

Sonification is the aural analog of visualization. Both provide the ability to find patterns in data and obtain information that may be outside the visual context of an editor, browser, or other textual display. A number of existing visual tools deal with a software program's structural elements. A representation of those tools is discussed in the following paragraphs.

Rational Rose [68] is representative of a class of tools that support the production and display of Unified Modeling Language (UML) diagrams [21]. Major UML diagrams supported by Rational Rose include class diagrams depicting classes and their relationships, sequence diagrams depicting calling sequence scenarios, and state diagrams depicting class or program states and their transitions. These diagrams provide orthogonal views of collections of structural elements in a software system. The diagrams are interrelated through an underlying model of the software system.

Sensalire and Ogao [140] provide a summary of ten software visualization tools ranging in purpose from UML support to graphical test coverage analyis. The typical tool summarized by Sensalire and Ogao is based on a node and arc paradigm. The nodes and links can be selected to provide additional, textual detail. Code Crawler, for example, provides a call graph in which the nodes vary by color, position, and size to indicate detail such as number of instance variables. Rigi [111] similarly builds a dependency model and provides dependency graphs. Tools such as Rigi provide containment and hierarchy information in the same diagram. Rigi provides multiple views, including the traditional node-link view and a hierarchical edge bundling view, the former emphasizing hierarchy while the latter flattens the hierarchy and emphasizes relationships. Storey's SHriMP [155] adds magnification of nodes of interest within several visual representations. SHriMP supports switching between top-down and bottom-up comprehension strategies by allowing zooming in and out to higher and lower-level views, respectively. Observation of

SHriMP revealed that views providing large amounts of data easily resulted in information overload, and that one particular view, the fisheye view (after Sarkar and Brown [134]), was rarely used as it did not particularly support a comprehension strategy. Creole [140] integrates SHriMP with Eclipse, providing radial, grid, and tree-map [143] views. SNIFF+ [155] was observed to support switching between systematic and as-needed strategies by keeping overview and detail views readily accessible.

Knight [84] demonstrated a novel visual means of exploring Java classes in which each class is a building in a cityscape. The height of each building represents the size of its corresponding class. Rooms within the building represent methods. The buildings are collected into villages representing larger structural collections of classes. Detail is expressed textually via signs found on the buildings and in the rooms. The tool's user can navigate to different classes and methods by virtually moving about the cityscape. The user can adjust the view by zooming in and out and observing from different visual angles.

Knight's cityscape illustrates the application of Shneiderman's visual information seeking mantra: overview first, zoom and filter, then details on demand [143]. One can gain an overview of the software system, then zoom to a given village and further zoom to an individual class or method.

The information collected by the tools described above is readily culled from the source code. Other tools cull and process information in ways that would require deeper search if done manually. One such class of tools supports concept assignment, the recognition and allocation of computational intent to corresponding implementation structures within the program's source code [14]. The HB-CAS concept assignment tool [56] was shown to have potential to reduce the cost of software module comprehension by alleviating the software maintainer of the need to summarize and abstract the module's concept list.

The visualization tools summarized in this section supplement a program's

source code by providing information in a more convenient or compact form. Sonification has similar potential. The attributes of structured sound, such as frequency and event duration, can be used in a denotational manner as can color, shape, and size. Chapter 3 provides an exposition of sonification and its capabilities.

2.5 Summary

This chapter has provided an exposition of software maintenance and program comprehension models.

There exist a number of classification schemes for types of software maintenance, grounded either in the intent of the maintainer or in the code itself. Program comprehension is essential for successful software maintenance. The maintainer forms a mental model of a program by using cognitive processes, existing knowledge, and other information structures. The program comprehension process may be top-down, bottom-up, or a hybrid of the two. Hypotheses are formed and verified, with beacons in the code aiding the verification process. The choice of programming language influences the comprehension process. Maintainers may intentionally choose a comprehension strategy, but such metacognition may reduce comprehension when working in an unfamiliar domain.

A key idea in program comprehension is that of concept assignment, in which recognized concepts are related to portions of the program, its operational context, and one another. Concepts may be delocalized in the code. Objects in an object-oriented language often encapsulate domain or design-specific concepts.

Sonification, as a tool for program comprehension, has parallels to visualization that give it promise. Sonification is discussed in Chapter 3.

Chapter 3

Sonification

3.1 Introduction

This chapter presents a literature review of auditory display and sonification relevant to this thesis. Particular attention is paid to the use of sound in the realm of software engineering. Non-speech sound is emphasized. A discussion of the rationale for non-speech sound over spoken text appears at the beginning of Section 3.5. Three advantages are included here.

- Humans process non-speech sounds differently and more quickly than spoken text, bypassing the language processing capabilities necessary for spoken text [41].
- A properly designed non-speech sound may be shorter in duration than its spoken counterpart [168].
- Non-speech sounds may be overlaid or expressed together in rapid sequence to further compress the time necessary to hear and process them [41].

Advances in auditory display and sonification have been made possible due to increases in computing speed and power, which in turn has allowed faster and richer application of digital audio sound generation and processing techniques.

Digital audio is treated in the book by Coulter [38]. Synthesis techniques are found in the books by Dodge [43] and Roads [131]. A treatment of the fundamental physics of sound is found in the venerable work by Jeans [73].

Section 3.2 of this chapter summarizes sonification and auditory display. Section 3.3 addresses sonification in computing disciplines. Section 3.4 discusses spatial and temporal metaphors and introduces a visual analogy. Section 3.5 discusses practices that have been abstracted over the course of studying sonification. Section 3.6 discusses human listening modes and learning sound associations. Section 3.7 summarizes the high-end audio engines and tools which make contemporary sophistication in sound possible. Section 3.8 discusses the previous applications of sonification to program comprehension. Finally, Section 3.9 summarizes the chapter.

3.2 Sonification and Auditory Display

The most common classifications of auditory display types in the literature are auditory icons [53], earcons [15], and data sonification [130]. Auditory icons and earcons are techniques for aurally communicating small, meaningful sets of information, as described below [53]. Data sonification is the use of sound to understand data sets [180].

Gaver [53] introduced the concept of *auditory icons* as "caricatures of naturally occurring sounds." Auditory icons are auditory analogues of visual icons: brief, unique, nonverbal sounds that represent objects. For example, the arrival of incoming e-mail might be represented by a metallic sound indicative of a mailbox. The metallic mailbox-like sound is more associative, and according to Gaver, more easily learned than the arbitrary sound of a beep or bell. A listener, according to Gaver, is conscious of the source of the sound, e.g. that it is a metallic object, rather than the parameters of the sound itself, e.g. pitch, making auditory icons similar to real-world listening experiences. An auditory icon may carry informa-

tion beyond the basic object representation. Gaver suggests, for example, that a distant mailbox sound, lower in amplitude and higher in reverberation, might be used if the e-mail window is hidden behind other foreground windows. Or, a lower-pitched metallic sound may indicate a larger incoming message.

As sounds exist in time and are transient, Gaver originally targeted auditory icons to represent events occurring in time in a computer system, complementing the more persistent visual representations. Gaver represented such events in his Sonic Finder, an extension of the visual Finder found in the Apple Macintosh [54]. Objects are represented as visual icons in the Finder; events that the objects are subject to are represented as auditory icons, exploiting the transient and temporal nature of sounds. The sonic representations are at various level of concreteness: dragging is represented by a scraping sound, indicative of physical dragging, while opening a file is represented as a "whooshing" sound. Auditory icons have been applied in other domains, such as vehicle collision systems [60]. Sodnik et. al. [147] found that automobile driving performance was better and perceived workload lower when using spatialized auditory icons instead of visual icons for secondary tasks.

Whereas an auditory icon is an unchanging sound or musical tidbit, the earcon is a brief, structured audio message. An *earcon* is usually a brief musical fragment, though non-musical earcons that extend auditory icons are possible [15]. The earcon adapts to represent some characteristic of an item in the represented domain.

A musically-based earcon is based on a primary unit called a *motive*, equivalent to a musical motive (of motif), a sequence of one or more pitches with a rhythmic stamp *earcon*. Related motives can be organized into *families*, larger groupings. Consider, for example, the two motives in Figure 3.1. They share the same rhythm, but the first is ascending in pitch while the second is descending. Variations in pitch, rhythm, volume, and timbre may be used to indicate differences in data.

Earcons need not be musically based, and in fact the class of sounds known as earcons can be considered to include auditory icons [112]. Earcons having this

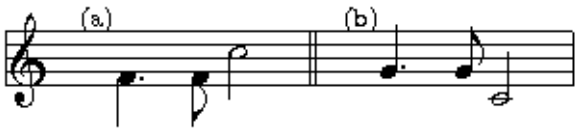

Figure 3.1: Earcons within a family

common rhythmic pattern can represent file operations. The upward pattern may indicate opening a file, while the downward pattern indicates closing a file. For deleting a file, the final note may be heard in conjunction with a metallic trash-can sound.

Audemes [46] are sound collages of two to five sounds layered or played in sequence over a few seconds. The individual sounds may be musical patterns, sound effects, infrequent sung words, or abstract noises. The collage can last from three to ten seconds, three to seven seconds being ideal. The audeme as a whole imparts an intended meaning by invoking an image from the listener's memory.

For example, consider the sound of a steam-powered locomotive, followed by horses' gallops along with gunshots. The image of a Western train robbery is invoked in those duly acculturated. This audeme may be used in a catalog for the visually impaired as a placeholder for longer narrative or other material about train robberies. Audemes are shown to work best when they are "richly metaphoric" implying that the sounds should be as concrete as possible. Sequences of audemes can be easily browsed, and due to their short length, both forward and backward navigation are possible. The Acoustic Edutainment Interface (AEDIN) test bed [46] was employed to demonstrate browsing capability by the sight impaired.

Mustonen notes that the classification of auditory signs into auditory icons and earcons is not compatible with the sign descriptions developed in the field of semiotics [112]. Mustonen uses the umbrella term *auditory sign* for auditory icons and earcons, under which the level of signification should be considered on

a continuum from iconic signs to symbolic signs. The latter employ arbitrary sign-object relationships defined by social convention.

It should be possible to represent software entities via auditory icons, earcons, and possibly audemes. The software entity's type (such as a class) might be recognizable by some sound characteristic, and its exact identification might be identifiable via a distinct sound pattern, either as a variation of the characteristic sound or a separately-heard sound. Characteristics of the entity might be represented by modifiers. Again, modifiers might be separate sounds appended to the main sound, or they might be variations in the main sound. Finally, sounds chosen from a consistent sound universe would help the listener by providing a retainable metaphor.

Data sonification involves mapping one or more parameters of sound to data values [180]. Some past applications have been sonified line graphs and sonification of large data sets [76]. Sonified line graphs exploit the temporal nature of sound by mapping the domain, or x value, of a line graph to time and the range, or y value, to pitch [28]. It has been demonstrated that two graphs can simultaneously be comprehended [28]. Tick marks and other features of the graph have been realized in sound [114]. While pitch is an intuitive range mapping for line graphs, it has been found that temporal mapping is better for box plots [122]. Data sonification can unfold in time much like a musical work. A data sonification depicting seasonal variation of Martian polar ice caps is an example [76]. Different orchestral timbres represent different parameters such as hydrogen concentration. Each timbre repeats continuously at varying rhythmic, density, or pitch levels to reflect changes over time in the data. The listener gains an overall impression of the state of the polar caps, which change over time. In such sonifications, the end-to-end listening time is compressed from months of actual time into minutes. Conversely, sonifications of millisecond-range event sequences are expanded into minute-range audio streams. The Martian polar ice cap sonification was applied

in an educational scenario to student understanding of the ice cap phenomena. Auditory, visual, and combined treatment groups performed equally in perceiving the ice cap data.

Data sonification has been applied for understanding very large data sets in an exploratory manner. Harding's geoscientific data investigation system (GDIS) combined visual, haptic, and audio representations of geological structures assembled from high-resolution bathymetric maps [63]. While the visual channel is considered primary, the secondary haptic channel allows a user to feel and hear surface features. The idea of simultaneous, multi-sensory investigation is expressed by the authors:

> One advantage of sonification is that the user's eyes are free to process visual data while hearing a different set of data. We have integrated a novel "sound map" into the visual rendering of surfaces, giving the user the ability to listen to a local surface property while simultaneously visually observing other properties. [63]

The GDIS investigation included a study of musical parameters that would best help listeners understand numerical data. Of pitch, timbre, and tempo, it was found that tempo was best perceived. The study demonstrated that, based primarily on tempo,

> . . . any subject, musical or not, can be trained to differentiate between five different audio signals and connect those audio signals with numbers 1-5 (a so-called ballpark setup).

The GDIS study did not consider durational sound aspects in general, which include both tempo and rhythm. The study suggests the possibility that durational aspects should be used to indicate quantity, notably where quantity is divided into discrete ranges. Static software measures such as size in lines of code per method and methods per class, which in exploratory situations are only roughly needed, can be represented in such a manner.

Childs and Pulkki sonified the movement and intensity of storms across the United States, compressing an actual period of months into an end-to-end sonification measured in minutes [33]. The user selects a geographical reference location to listen from. Storms approach, occur, and depart as storm-like sounds in audio space according to a dome-like model. Listeners were able to detect patterns in the occurrences of storms. The effective use of distance and direction in audio space for a successful sonification is particularly promising. Polli, from a film and media background, also sonified storms, concentrating on characteristics of individual storms [127]. Different variables were represented using a variety of sounds: vocal sounds, instrumental sounds, insects and other environmental sounds. Percussive sounds represented water-related variables; long tones represented pressure and temperature related variables. The storm thus became a five-minute or so musical composition. Decisions were made such as translating atmospheric pressure to a very low frequency sound. In doing so, "listeners lost the ability to hear a detailed melody line describing the pressure changes, but gained a visceral sense of the storm."

Interactive Sonification

Interactive sonification can be aimed in two directions: for the sighted and for the visually impaired. The sighted can use visual navigation techniques, reserving sound for information presentation where it reinforces visual means or provides some advantage over it. The visually impaired will navigate using sound. GDIS, described above, is clearly for sighted users, as it contains interacting visual, aural, and haptic components. The interactive sonifications described below are targeted primarily for visually impaired users and can be used while working in an exploratory context.

Zhao, Plaisant, and Shneiderman proposed an Auditory Information Seeking Principle (AISP) for exploring a general data collection in an interactive, visual

environment [185]. It is an adaptation to the audio domain of Shneiderman's Visual Information Seeking Mantra [142]. AISP exploration occurs in four phases: gist, navigate, filter, and details on demand, mirroring the phases in the Visual Information Seeking Mantra. In the visual domain,

1. *Gist* provides an overview at a glance of a data collection's pattern. it serves as a guide for exploration. Anomalous data can be easily detected in contrast to the overall pattern.

2. *Navigate* allows the user to quickly visit different parts of the data collection, selecting interesting areas of the data.

3. *Filter* allows the user to focus in on data with desirable characteristics, eliminating unwanted data from consideration.

4. *Details on demand* allows the user to select a subset of data or a single item and receive additional information about it.

In the visual world, Shneiderman's Visual Query concept is built around the Visual Information Seeking Mantra. One of its applications is as a real estate locator, the DC HomeFinder [142]. Available houses for sale appear as points on a geographic city grid, providing a gist. The user may zoom to and visit different areas of the city, providing navigation and a more localized view of the data. Sliders allow the user to filter data by price range, number of rooms, and other features, reducing the number of points in a geographic area. Finally, the user can select an individual house to obtain its full detail. The user can iteratively return to a gist and select other houses. Visual navigation, filtering, and details on demand are designed to operate as fast as possible, in the range of 100 milliseconds or less per operation, giving the user quick feedback and the feeling of real-time control of the tool. This apparently real-time way to visually explore data is known as a *dynamic query*.

Zhao based AISP's principles for interactive exploration of information spaces using sound upon the principles of the Visual Information Seeking Mantra. A

geo-referenced statistical data set was sonified for access by the visually impaired. Again, the phases are gist, navigate, filter, and details on demand. The 100 millisecond requirement is discarded, as sound itself is a time-dependent medium, one coherent sound or group of sounds requiring far more than 100 milliseconds. The AISP phases each meet certain requirements:

1. Gist, being processed in short-term memory, should itself be in the approximate range of five to thirty seconds in length. Gist can be a serialization of the data items. If so, there needs to be a scheme to hear all the items in the gist within the desired time frame. If the data set is large, aggregations must be used.

2. Navigation should allow the user to play, pause, resume, and rewind through the data in the gist, initiate zoom and selection, and receive feedback to what extent actions have been performed.

3. Filtering requires operations well outside the 100 millisecond range, possibly involving some audio process independent of gist and navigation and performed while paused.

4. Even when the gist is achieved through non-speech sound, details on demand may effectively make use of speech. Otherwise, there may be too many non-speech sound mappings to remember.

The geo-referenced data set is sonified in two ways: as an enhanced table and as a spatial choropleth map. In the enhanced table, gist is an ordered serialization of the fifty U.S. states plus the District of Columbia. Ordering is a mapping from west to east and north to south to the single time dimension. The state name is spoken along with a 200 millisecond pitch indicating a value, such as the state's elevation. The lowest pitch corresponds to middle C (approx. 261 Hz), with higher value mapped to higher pitch. Navigation and detail on demand are keyboard-oriented, the assumption being that a visually impaired user, once oriented to a

keyboard, can remain oriented. Filtering does not appear to be implemented. The table is played as a stereo image but without left-right panning.

In the spatial choropleth map, the sound's azimuth and altitude are respectively mapped to east-west and north-south state location. Azimuth is from -90 degrees to 90 degrees (where zero degrees is front and center). Altitude is -31 degrees to 63 degrees (where zero is level). Sound localization is achieved using a generic head-relative transfer function (HRTF) [43] to achieve binaural sound through headphones. Keyboard-based details on demand is also used. In the gist, a 200 millisecond tone indicating which value is to be heard is played, followed by a 100 millisecond tone representing the value, both at the state's mapped spatial location. Keyboard-based commands allow the user to navigate left-right and up-down.

Zhao reports that there is evidence from the pilot study that AISP is in line with users' pattern recognition strategies. Both the enhanced table and the spatial choropleth map were shown to be effective for conveying the geo-referenced data. Subjects were able to identify patterns in the data.

To provide gist in an auditory environment, Zhao found it important to provide serialization of the sounds. Pauses of ample length occur between items: 100 milliseconds between columns in the choropleth map, and 1/2 second at the end of a row in the map, each of which follow sound patterns in the 300 millisecond range representing simple data values. In an environment such as a software IDE, in which each represented item covers a more complex range of information, the sound patterns would be longer, and the pauses would have to be at least as long as the row-end pause of 1/2 second.

Zhao also reports that the ability of subjects to locate the sound sources in space was inaccurate. Improved accuracy would require improved HRTFs and expensive head tracking devices. Accordingly, localization in a software IDE, while usable, should not be depended upon as a primary differentiator of items or

their characteristics.

Kildal and Brewster created a tool to support interactive, audio data tables [80]. Each table cell contained a numeric value, represented by a piano-like sound whose pitch was mapped to the value. Users can explore the table sonically using a 2D touchpad. The user can traverse rows, columns, or diagonal paths through the table. The user can also play the sounds for a row or column in sequence or overlapped without the need to physically traverse the row or column. This would provide a quick idea of the range of values and their weighting. Experimental subjects successfully garnered patterns in the data through sonic exploration of the table.

The idea of scanning over cells of a table and hearing a sound associated with each cell is reminiscent of Shneiderman's visual queries, giving instant feedback and the impression of real-time control. Fast scanning may not work as well in an IDE, in which the sounds are anticipated to be longer and more complex than a simple tone, but slower scanning or hovering can be implemented. Faster scanning would result in silence to avoid confusion, time lag, and overload of the auditory sense.

3.3 Sonification in the Computing World

Sonification has been applied in the computing realm for run-time monitoring of algorithms [48] and computer networks [55]. Run-time monitoring is a natural application for auditory display and sonification, as sound itself is temporal in nature and able to interrupt and re-focus the human attention stream. Monitoring can, in general, utilize sound actually produced by the monitored source, as in stethoscope monitoring of one's heart. Computer algorithms and network traffic produce no sound of their own, so recorded or generated sounds are mapped to source events and values.

The Zeus algorithm animation system provided sound as well as visual views

to monitor sequential and parallel algorithms as they ran [30]. Each comparison or data movement in a sorting algorithm resulted in a tone whose pitch was mapped to the element's value. Because a sorting algorithm is highly iterative, the sequence of pitches produced a distinctive signature. Mode changes can be detected via changes in the signature. Some data relationships can be detected that did not manifest themselves through any of the visual views. In addition to pitched sounds, Zeus employed auditory icons, notably when a value inserted into a hashing algorithm resulted in a collision, in which case a "violent car crash" was heard [29].

Francioni, Albright, and Jackson sonified the execution behavior of algorithms running on parallel processors communicating via shared memory [48]. Simple musical tones and short melodies were mapped to program execution in three different ways. In the first mapping, each processor was given a distinct timbre, and distinct notes were used for send, receive, and pending events. A sustained tone signaling a pending event served to connect a send event and its eventual receive event. Send, pending, and receive events were mapped to different stereo positions for aural reinforcement. Listeners can detect send events that were never received, as the unconstrained sustained tone violated the normal pattern. Events which occurred simultaneously on multiple processors were serialized, an adjustment to ensure that events were not aurally lost. The second mapping depicted busy versus idle processors. Each idle period of significant duration was signaled by a bell, followed by a lower-intensity string-like sound, its amplitude increasing with the idle period's increasing duration. The bell and string-like sound appeared at a unique pitch for each processor. Listeners can detect significant idle periods for different processors. This mapping can be used to tune parallel operation to reduce idle time. The final mapping depicted flow of control within each processor, different pitches represent different types of events. Bottlenecks in flow of control can be detected via the absence of a pattern, demonstrating that

conveying information by negative means (absence of sound) is viable.

The bell and string-like sound, together serving as an earcon, are particularly intriguing. The bell signals an event, and the string-like sound then highlights the length of that event. Stated differently, the bell represents the event itself, and the string-like sound is a modifier that provides additional information.

Personal Webmelody was a musical sonification of web server activity [5]. Short, system-generated music patterns represented web server events. Audio monitoring can be intermixed with external audio files such that the listener's preferred music is played while the real-time status of the web server can be monitored. The listener's preferred music does play into the sonification; if a web server goes down, silence replaces the music, signaling and reinforcing the condition through *lack* of sound.

A particularly impressive sonification vehicle is the Peep Network Auralizer for real-time monitoring of computer networks [55]. Sounds were selected for employment from an environmental sound universe - birds, crickets, waterfalls, and other sounds in a forest-like setting. The sounds were recorded from nature. The researchers felt that different sounds from the same natural setting would be heard as concordant by the listener, analogous to musical chords comprised of concordant pitches. Sounds represent three basic categories of network occurrences: *events*, *states*, and *heartbeats*. An event is something that occurs once, such as an incoming e-mail message arriving on a server, represented by a single bird chirp. As incoming and outgoing e-mail messages often occur in pairs, their chirps are complementary call and response chirps. Should outgoing e-mail suddenly stop, the network engineer would notice the absence of response chirps as anomalous. States are quantities measured during continuous operation of the network, such as number of users. States are represented by continuous sounds like wind or a waterfall, which progresses from quiet to loud as more users join the network. The network engineer, like someone relaxing at a waterfall, can generally

ignore its sound. When the waterfall becomes too loud, it attracts the network engineer's attention. Heartbeats are sounds that occur at constant intervals, and they either represent quantities or presence of an operating component. Sparse cricket chirping may represent low network load, while dense chirping represents high load.

Peep demonstrates that listeners can garner quantitative, parallel information through superimposed sound patterns. It also demonstrates that sounds from a consistent universe provide a learnable and retainable metaphor. Finally, Peep demonstrates that complex sounds, in this case sounds from the natural world, work well for auditory display.

3.4 Spatial, Temporal, and Visual Metaphors

Pinker [126] describes evidence, gained from observation of spoken and written language, that the human mind categorizes matter and temporal occurrences similarly. Pinker states that,

> ...the mind categorizes matter into discrete things (like *a sausage*) and continuous stuff (like *meat*), and it similarly categorizes time into discrete events (like *to cross the street*) and continuous activities (like *to stroll*). [126]

Hence, it would appear natural to represent a collection of objects, especially but not necessarily ordered objects, as a collection of events in time, or vice versa. Indeed, precedents exist in the auditory display of software entities. Objects in relational diagrams have been aurally depicted as sound events separated in time, connected in time by a sound representing the relation between the objects. Experimental subjects have been able to reconstruct relationships between objects by listening to the aural depiction. The objects are unordered [107]. A sonification of the London Underground map is constructed in a similar manner with ordered

subsets of objects [115].

Pinker proceeds to describe a mental "zoom lens" which allows a single object to be broken down into a collection of objects and, more importantly for the present purpose, a complex temporal event to be broken down into constituent events. The zoom lens analogy suggests that audio representations may be constructed so that the listener can determine the meaning of a coarse event and, if desired, also determine the meaning of finer subordinate events, as long as they are seen as belonging to the coarse event. In this manner, internal zoom on the listener's part replaces explicit, interactive zoom employed by Metatla, with a slightly higher cognitive cost.

The use of box and line diagrams [64] to represent software constructs suggests an approximate analogy in the use of abstract sound structures to represent software constructs, as shown in Table 3.1. Boxes and lines are abstract, as are sound

Visual	**Aural**
pixel	vibration
symbol (pattern of pixels)	note or event (pattern of vibrations)
complex symbol (e.g. box and its partitions)	complex event (e.g. chord)
multiple symbols	multiple complex events

Table 3.1: Box and line analogy, in increasing complexity

events such as musical notes. Neither presents an intuitive association to software constructs, which may themselves be abstract. Boxes, lines, and other shapes combine in Unified Modeling Language (UML) diagrams [21] to partly describe classes and other software constructs. Musical events can also be so combined, both sequentially and serially. They, too, will only partly describe each software construct, as "software entities are more complex for their size than perhaps any other human construct," making them "differ profoundly from computers, buildings, or automobiles, where repeated elements abound," per Brooks [26], who indicates that design diagrams do not necessarily get to the "essence" of a software

construct. Brooks continues,

> *Likewise, a scaling-up of a software entity is not merely a repetition of the same elements in larger sizes, it is necessarily an increase in the number of different elements.*

UML accomplishes scaling-up through many repetitions of boxes and lines, each having different contents. The aural analogy is many collections of events having the same structure but different sounds.

The box and line analogy is necessarily inexact, as the visual medium exists in two physical dimensions while the aural medium exists in one temporal dimension. The analogy is also constraining, as sounds have high expressive potential, while the shapes encountered in box and line diagrams are neutral.

3.5 Design Guidelines

There is evidence that non-speech sound is processed differently by humans than spoken text, faster in some situations. Leplâtre and Brewster found that a group of subjects using a telephone menu containing earcons performed required 17% less key presses than a control group using a text-only menu, and it also decreased the number of errors [90]. Vargas and Anderson reported similar results (15%) with an automobile control panel simulator, but they noted that the earcons were longer in duration than equivalent spoken text in their setting, and accordingly took longer to process [168]. Their findings indicates that care must be taken in the design of earcons and their placement. An earcon should ideally communicate equivalent or more information than its spoken counterpart within the same duration.

Brewster, Wright, and Edwards made explicit the observation that earcons should be of short duration. In their guidelines for earcon creation, they state, "Earcons should be kept as short as possible so that they can keep up with interactions in the interface being sonified [25]." Other guidelines from the same

source are useful:

- *When designing a family of earcons start with timbre, register, and rhythm.* Suppose we elect to represent a particular software entity, say a given "accessor," with a double bell-like sound in a medium-high register. All accessors should then be represented by double (or at least multiple) bell-like sounds in a medium-high register, individually varying by their exact pitches and other factors that retain the characteristic timbre, register, and rhythm.

- *The maximum pitch used should be no higher than 5kHz and no lower than 125Hz to 150Hz.* Pitches lower than the recommended range are easily masked by other sounds, and pitches higher than the recommended range may not be easily heard, especially by but not limited to older listeners.

- To make an earcon capture the listener's attention, increased intensity (amplitude) is effective but crude. A variety of other techniques is available, including use of accentuated rhythm, or even atonal or arhythmic sounds. One can envision the best of both worlds: using the atonal and arhythmic sound of a door closing, say to represent a *close* method, immediately followed by a musical motif indicating some characteristic of the method. This foreshadows the next recommendation,

- *Compound earcons.* While a 0.1 second gap between successive serial earcons is recommended, we envision that there need be no gap if the first earcon fades out substantially before the second begins.

- *Spatial location.* A suggestion is that each family of earcons be assigned a distinct spatial location. However, one can envision an overriding use for spatial location, namely to separate represented items by providing each item a location in space. The metaphor of items in space is closely tied to sounds in space.

Frauenberger devised a design-pattern methodology for capturing best practices from the auditory display community [49]. His methodology is an adaptation of the widely used design pattern methodology in software engineering [52], and it is partially based on prior work by Barrass [6]. Frauenberger collected an initial set of design patterns to illustrate and validate the methodology. Each pattern poses an audio display problem and captures known solutions. Patterns applicable to the problem at hand are discussed in the following paragraphs.

- *"Interaction design exploiting auditory means can impose increased cognitive effort on users. This results in users perceiving auditory displays as annoying or tiring."* The pattern points specifically to monotony experienced while using an auditory menu system using that employed repetitive speech to indicate the position of items in the menu [50]. Providing sounds from a rich, non-speech sound universe should alleviate the monotony of repeated speech. The user should have the ability to turn the sound off entirely, especially when performing tasks other than the sound-appropriate ones. The user should also have control over which groups of items are heard.

- *"The user should have easy control over . . . how long it takes to explore the data [in a large table] . . . Instead of looping through the chosen time-line in a data set at a constant rate, provide interactive control for the user to change the speed of the presentation. This allows users to explore the data value for value or skim through the data quickly omitting much of the detail, but gaining overview."* The file explorer or class browser in an IDE are essentially data tables. In an exploratory mode, the user will use a pointing device such as a mouse to click or hover over software entities such as classes and methods. The sound should be heard as the user clicks or hovers over each item. The user may stop on individual items or progress through the entities at his or her own rate. In a multi-modal environment that includes both visual and audio presentation, non-speech sounds may be used for

overview while spoken text is be used for detail.

- *"A directional link between entities has to be expressed by auditory means."* In his audio UML diagrams [21], Metatla represents an arrow by a long tone followed by a short tone, audio equivalents to the line and arrowhead comprising a visual arrow [107]. In a multi-modal IDE, one may focus on an entity of interest and request to hear: only those other entities that call or are called by the entity of interest; those entities from which the entity of interest inherits; those entities which inherit from the entity of interest; or some other relationship between entities. Knowing what is to be heard, it may be possible to omit any explicit representation of a link. If the link is necessary, Metatla's audio arrow may be used.

Norman informs auditory design possibilities through his introduction of constraints and affordances to visual, user-interface design [116]. Constraints are associations or concepts which are well known due to habits and cultural conventions. New user interface concepts that are similar to something already well known are easily learned as they conform to cultural constraints. Hence the concept of a "window" in a graphical user interface. A sound issued to announce a remotely-initiated instant messaging session may resemble a telephone ringing or a door bell. An error, on the other hand, is unlikely to be announced by those sounds. Affordances, closely related to physical constraints, are natural properties of objects that suggest a type of interaction. For example, a flat rectangular area in an otherwise vertical shower wall suggests that one should place soap there.

Vickers has advocated increased awareness and application of aesthetics on the part of sonification designers [169]. Aesthetic considerations range from improving the realism of the sounds and the virtual acoustical environment to intermixing a variety of musical and non-musical elements, to the point that one cannot classify a sonification as musical or non-musical. Vickers suggests that designing for aesthetics, especially in the realm of tonal music and musical fragments, would not make

the sound scheme a language that the listener would have to learn, and therefore would not be restricted to or performed better by the more music-literate listeners. If anything, cultural differences might determine variances in understanding based on the sound scheme. He also argues for relaxation of familiar (19^{th} Century) tonal and structural conventions in defining a musical sonification, defining music as "organized sound," which is neither tonal nor atonal but may possibly utilize an eclectic mix of tonal music, atonal music, sound collage sequences as in *musique concrete* [78], and other types of sound, from both acoustic and electro-acoustic sound sources. Going a step further, Vickers questions the distinction between sonification and musical composition as a matter of one's perspective.

3.6 Listening, Processing, and Learning

Tuuri, Mustonen, and Pirhonen have built upon work in psychoacoustics to differentiate among listening modes [112][167]. Tuuri, Mustonen and Pirhonen agree with Gaver, above, that everyday listening is focused on sound-source actions and events rather than on conscious evaluation of the sounds themselves. There exist, however, *acousmatic* situations, in which the action or event causing the sound is hidden from the listener, resulting in *reduced* listening that focuses on the characteristics of the sound. There are also situations in which sounds are ambiguous, causing the listener to invoke contextual information to aid in discrimination and interpretation.

Tuuri summarizes six activating systems which participate in sound discrimination and interpretation: reflexive, denotative, connotative, associative, empathetic, and critical. The reflexive system provides fast, pre-conscious responses of a physiological nature, requiring little cognitive processing. The critical system provides "reflective self-monitoring concerning the verification of perception and the appropriateness of one's responses," requiring significant audio processing. The other systems require intermediate degrees of cognitive processing. The six

activating systems work alone or in combination to effect the processing of sounds in eight modes, shown in Table 3.2, ordered from least to highest processing load.

mode	**type**	**description**
Reflexive	pre-conscious	attention-focusing, startle response
Connotative	pre-conscious	immediately invoked, free-form connotation
Causal	source-oriented	determining likely cause of the sound
Empathetic	source-oriented	determining likely emotional state of source
Functional	context-oriented	determining purpose of the sound
Semantic	context-oriented	determining symbolic/conventional meaning
Critical	context-oriented	determining suitability of sound for situation
Reduced	quality-oriented	describing the properties of the sound

Table 3.2: Listening Modes, adapted from Tuuri

To keep cognitive overhead low, it is reasonable to expect sounds which stimulate reflexive listening to be optimal for warning situations and sounds of a connotative nature to be optimal for iconic use. It is also reasonable to expect sounds in an emotion-free understanding context to be designed such that they are not listened to in an empathetic mode. Sound and sound patterns serving as arbitrary signs cannot be truly selected at random, as they may give rise to conflicts in one or more processing modes and therefore serve as *auditory distractors*. It may be necessary to listen to abstract, arbitrary-sign sounds in a reflexive manner when first learning their associations, but once committed to memory, reflexive listening should no longer be necessary.

Card, Moran, and Newell combined research in cognitive psychology with computing principles to formulate the Model Human Processor, a simplified model of human perception, processing, decision making and action applicable to human-computer interaction [31]. In the model, auditory information is perceived within

a 50 to 200 millisecond time frame, chunked into information units, and placed in a short-term auditory image store, where it resides for 900 to 3500 milliseconds, as contrasted with short-term visual image storage of 70 to 1000 milliseconds. The auditory image store is within working memory, which is capable of handling approximately seven simultaneous chunks of information, in agreement with Miller's seven plus or minus two principle [110]. Learning occurs as information chunks are committed to long-term memory. Selective retrieval from long-term memory improves access time and reinforces learning.

The Model Human Processor incorporates a number of operational principles, including the *Discrimination Principle*, which states that the difficulty of recalling an item is increased if there are similar items in memory, as recall is cue-based. Items in working memory are most easily confused with other items with similar acoustical properties, while items in long-term memory are more sensitive to items with similar meanings. Also notable is the *Power Law of Practice*, which states that perceptual-motor learning improves task performance time on successive trials through a power law. On the other hand, information in long-term memory is lost during long periods without rehearsal such as overnight.

The work of Card, Moran, and Newell suggests that an audio representation of a particular software item or class of items should be well bounded temporally, with enough time afterward for cognitive processing and commitment to long-term memory. Each item or class of items should be represented by sounds or sound patterns acoustically distinct enough to differentiate them from others on recall. If audio metaphors are used, their meanings should also be suitably distinct. Training and usage scenarios should include sufficient repetition to bring task performance times to a near-minimum. Training should be reinforced immediately prior to any experimental trials to restore information lost during long periods without rehearsal.

Stephan et. al. [154] studied learning and retention of associations between

auditory icons and referent events, discriminating among direct, related, and unrelated associations. Learning and retention were high irrespective of association type. Direct and strong indirect associations were better retained in the short term. Additionally, direct associations were better retained than indirect associations on follow-up four weeks later. The results suggest that direct associations, followed by strong indirect associations, should be preferred over unrelated associations, but that all types are acceptable. Lucas [97] found that association accuracy of musical earcons is improved when study subjects are presented with their structure, having implications for training.

3.7 Audio Engines and Tools

High-end audio engines support real-time generation and processing of sophisticated digital sound. MAX/MSP, PD, SuperCollider, and Csound are the most widely used and well-tested such engines [24][39][105][186]. In addition, the Audacity audio editor is usable for high-end manipulation of audio streams [3].

MAX/MSP and PD are intended for real-time processing of internal and external audio sources. MAX/MSP is an audio engine originally developed for the Apple Macintosh [39]. PD is a similar, open-source audio engine, developed by Miller Puckette, the creator of MAX/MSP, who considers MAX/MSP and PD to both use the MAX family of graphical languages [186]. Sounds and sound patterns are obtained by building a graphical network of objects, a virtual analog of the collection of patches an analog music synthesizer or, more loosely, a telephone switchboard. Event objects such as sequencers provide time-regulated successions of musical events. The final stages in the network are output objects that send the sound to the computer sound card or other device.

SuperCollider is a real-time audio engine and an audio synthesis programming language [105][161]. Written by James McCartney, it runs on MacOS, Linux, and Windows. It supports plugins, extensions, and programmer-provided sound ob-

jects. Its real-time, interpreted programming language, reminiscent of Smalltalk, includes unary operators, binary operators, oscillators, noise sources, filters, controls such as gates and latches, amplitude operators, delays, FFTs, sampling and input/output, event control, and miscellaneous classes such as mixdown units. The real-time language serves as a network client to the sound synthesis engine, communicating via the Open Sound Control (OSC) data transport protocol for musical events [118]. SuperCollider is introduced more fully in a 2008 Linux Journal article [125].

Csound is a high-end software engine for creating and processing digital audio [24]. It is a direct descendant of the earliest score-based computer music systems developed by Max Matthews and dating back to 1969 [102]. Unlike MAX-family tools, Csound uses textual input, separating its input sources into an *orchestra* file and a *score* file. The orchestra file contains "instruments" built upon a language reminiscent of a software assembly language. Each line of text utilizes commands which invoke sound generation and processing objects implemented in C. The objects are connected textually much as MAX-family tools connect them graphically. The score file contains performance instructions, turning on and off instruments and specifying the durations of notes and other performance parameters as defined by each instrument. Real-time score input can be substituted for a static score file. MIDI control is also available [109].

MAX/MSP, PD, SuperCollider, and Csound can each be integrated with an IDE such as Eclipse. Output from Eclipse would serve as input to the chosen audio engine in real time via an interface using Unix pipes, TCP/IP sockets, or a similar technique. The audio engine can run on the same machine as the IDE, or if necessary it can run on a separate machine to take advantage of extra processing power. Csound offers several advantages over MAX/MSP and PD for Eclipse integration. The user should not have any need to directly control the sound engine, so the MAX-family's graphical, network-based interface is unnecessary. MAX/MSP

and PD provide the intuitive graphical ability to produce sound in continuous loops, but that ability was not considered necessary to realize the envisaged aural concepts. Freely-available Csound instrument libraries provide a wealth of instruments. Finally, Csound's ability to accept real-time score input from Eclipse has been demonstrated [12]. Csound's main advantages over SuperCollider are the richness and flexibility of its as-distributed sound library and, for this particular project, familiarity by the researcher.

Audacity is an open-source audio editing tool [3]. Audacity displays sound as a waveform, showing amplitude versus time. Audacity can be used to provide editing of sounds serving as input to Csound, to bound them, eliminate noise, and otherwise clean them up or preprocess them. Audacity can also be used to prepare audio streams by pasting in successive sounds. Notably, it was used to prepare the audio stream employed in the experiment described in the following chapter.

3.8 Sonification for Program Comprehension

Most program sonification research to date has addressed the dynamic, or behavioral, aspects of computer programs at the program statement level. It has been reported that, as early as the 1950's programmers used AM radios to listen to the interference caused by computers, monitoring the CPU and recognizing, to some degree, improper behavior [171]. Later, sound was proposed to enhance program visualization, transitioning in some cases to a vehicle in its own right.

InfoSound allowed software developers to create and assign sound effects and musical fragments to chosen program statements, then listen to them in a continuous stream during program execution [150]. Using InfoSound, the developers can detect successions of events that were difficult to detect visually. The Auditory Domain Specification Language (ADSL) enabled listeners to specify which program constructs would be sonified and with which simple sound each would be represented [19]. Each type of construct resulted in a "track;" there would be a

track for loop constructs, another for boolean evaluation of expressions, etc. The listener would select tracks to play, then hear them in program execution sequence.

The visual programming language Sonnet, developed for the aural monitoring and debugging of an executing program, eventually became a platform for more general real-time investigation [72]. One builds an event-triggered sound processing network in a style similar to that of MAX [39] described in Section 3.7 below. Again, tagged program statements trigger sounds, and program execution can be heard as an audio stream. Similarly, the LISTEN system, with its non-visual programming language LSL, can be used to tag program statements with simple sounds and musical tones, then hear the program's execution [16]. Audio execution traces of bubble sort and selection sort algorithms are available at Mathur [101]. The percussive sounds and tones heard allow one to obtain a sense of the sort algorithm. However, lengthy exposure is reminiscent of listening to a machine, and it may result in loss of attention. Baecker, DiGiano and Marcus designed a visualization system for debugging that incorporated audio elements similar to those in the sonifications described above. [4] It was employed to help undergraduates understand algorithms.

Alty and Vickers developed the CAITLIN system and used it to evaluate the effectiveness of audio display for dynamic program comprehension [170][171]. CAITLIN employed musical motifs, much like the character-identifying motifs in Prokoffeiv's *Peter and the Wolf* and Richard Wagner's operatic leitmotifs [61]. A different motif was applied to each point of interest in a Pascal program, such as the beginning of an IF statement or evaluation of a Boolean expression inside it. Each motif was a single melodic line performed using a unique timbre. Constructs having a significant duration, such as an IF statement, were continued with a continuous drone tone until their end. Thus an IF statement would have an opening motif, a Boolean evaluation motif, possibly an ELSE statement motif, and a drone terminated by a closing motif. The sounds were realized by an electronic

musical instrument receiving MIDI input from CAITLIN. Unlike its predecessors, CAITLIN made use of fixed musical motifs designed to work together to form a consistent, harmonic scheme that is primarily consonant. This allows a longer and more pleasing listening experience than do the LISTEN sort examples, and it affords clearer understanding overlapping sounds. In an experiment, twenty-two undergraduate computer science students were asked to use CAITLIN to identify sonified constructs which were not placed in a meaningful context. Sequential and nested constructs were included. Results were marginal. The students were then asked to perform eight debugging exercises by listening to the constructs in context. Each exercise contained branching errors in the program flow. The errors were covert: the program ran to completion but provided output other than that expected. The subjects found 60 of 88 sonified bugs (68%) and 46 of 88 non-sonified bugs (52%). Thus, the students were demonstrably able to find about half of the bugs through sonification, while an unsonified treatment yielded significantly better results. Time to find sonified versus non-sonified bugs was not significant, and level of musical experience was not significant. Evidence was found to suggest that the effectiveness of the sonified treatment increases with the cyclomatic complexity of the program.

Finlayson delved into the realm of static program comprehension with the spoken and non-speech AudioView [47]. In the spoken version, the listener was told what program structures were encountered in Java source code and how many statements resided in code blocks, including those within IF and FOR statements. In the non-speech version, the spoken text was replaced by earcons. Variations among earcons included rhythm differences and musical timbres from different families of instruments. Pitch was not used as a discriminator. A quantitatiave study was conducted in which subjects were presented with speech, non-speech, and combination AudioViews of source code fragments with and without errors. Subjects were asked to identify code by hearing an AudioView and detect errors.

Results showed that, at the statement level, the non-speech AudioViews were less accurate than the spoken AudioViews. However, the results are only suggestive and admittedly "early," as the sample of six subjects was small and the earcons seem similar to one another in terms of pitch and rhythm.

Berman and Gallagher developed three techniques to sonify program slices [11]. A *forward program slice* with respect to a statement of interest, or *slice point*, is the set of source code in the program that depends upon that statement [178]. Conversely, the slice point depends upon the code that is in a *backward slice*. A slice may further depend on specific variables at the slice point. In the first technique, individual program statements are heard as single pitches. Variation in pitch is used not as a differentiation device, but as a repeating pattern to avoid monotony when listening to a large number of statements and emphasize breaks in the pattern. In the second technique, an entire method is heard as a cluster of pitches, giving an impression of the number of statements in the method that belong to the slice. The third technique makes use of granular synthesis [43]. For a given Java class, the listener hears a signature sound cloud consisting of numerous sound grains distributed according to parameters fixed over the cloud's duration. The size of the class corresponds to the overall pitch range of the cloud, and the percentage of the object's code belonging to the slice corresponds to the cloud's density of grains. The cloud can be heard as background along with one of the other two techniques in the foreground for methods within the class. Applied to a project browsing environment, the cloud sound changes as the listener progresses between object boundaries while hovering over different methods. An informal, qualitative study revealed that the third technique had the advantage of lasting any time span, controlled by the listener, so that it may be analyzed over time without having to be replayed. The slice sonifications demonstrate that musical yet non-melodic and non-harmonic sounds can be employed to represent differentiating and quantitative information about program source code.

Boccuzzo and Gall adopted a hybrid approach in which spoken text and non-speech sound support and extend visualization of software metrics [17]. Visually, a software unit is represented by a house whose size, shape, and color are mapped to software metrics, making the representation a *cognitive glyph*. For example, the height of the roof represents the number of lines of code. Sounds can provide notification, indicating outliers such as an unusually high number of critical bugs. Sounds can also serve as data-driven cognitive glyphs in their own right. To do so, a tone is presented whose duration, loudness, sharpness, pitch, roughness, and oscillation map to various metrics.[1] Boccuzzo and Gall found, in a human-subject study, that duration and loudness were hard to perceive in non-ideal environments, and loudness in particular was hard to map linearly. A more successful strategy was to map tones having discrete differences to ranges of values so the listener can get a rough idea of the value. In subsequent research, Boccuzzo and Gall supplemented exploration of units shown in a software visualization with an "ambient" sound similar to that of bubbling [18]. The character of the sound changes as the user explores different units in the visualization, allowing the user to hear metrics such as the number and size of changes since the last release.

Several parties have recently extended IDE's to provide auditory display, supplementing information available visually. One extended Microsoft's Visual Studio while the other extended Eclipse.

Stefik and Gellenbeck extended Visual Studio's run-time debugger with spoken text [153]. Their Sonified Omniscient Debugger (SOD) builds upon their own previous work, in which a Visual Studio extension was known as the Wicked Audio Debugger [152]. SOD and its predecessor operate at the source statement level, announcing variable and array value replacements, loop iterations, and statement nesting, as well as values and memory addresses of variables and array elements navigated to in a list of active variables. For example, it might say, "v sub zero

[1]The listed characteristics are called the *Zwicker parameters* in psychoacoustics [187].

integer of value 5" to announce that the user has navigated to integer array element zero having a value of 5.

SOD usage was evaluated with the debugger in purely visual mode, purely auditory mode, and a multimedia mode with both visual and spoken cues. Accuracy and performance time data on debugging tasks performed by forty student subjects, mainly undergraduates, was collected. Task performance time was shortest with visual-only presentation, marginally longer with multimedia, and significantly longer with audio-only. Performance time improved as the subject moved through the different presentations, performing one task using each medium, indicating a learning effect independent of medium. Accuracy as measured by a comprehension score did not significantly vary among the three media.

Stefik and Gellenbeck found that subjects who performed tasks using multimedia mode last performed better in that mode than did others for whom it was first or second. They attribute that particular performance improvement to a learning effect, the subject having already had experience in each mode, and no subject having ever previously been exposed to any auditory mode. They conclude that a learning effect took place, and therefore that auditory cues do not afford *instant usability*. They also observe that newcomers to the environments that include audio tend to listen to all auditory cues, only later learning to separate those they need from those they don't.

Subjects in the Stefik and Gellenbeck's experiment were given ten minutes training time followed by performance of the experimental tasks. Learning effects may be mitigated in similar experiments by

- extending the training time, possibly by reducing the number of experimental tasks to keep session time about 90 minutes,
- providing additional offsite training prior to the session,
- providing redundancy in training,

- providing practice tasks prior to the experimental tasks.

Hussein, Tilevich, and Bukvic extended the Eclipse IDE with an auditory display, using MAX/MSP as their sound generator [67]. They conducted an initial experiment to determine quantitatively that non-speech sonification can be effective in a program comprehension setting. Static program information of a numerical nature was sonified: number of lines of code within a method, total number of method calls within a method, and number of calls to a given API by a method. The ability to know such measures may help a maintainer to strategize during debugging or help a quality assurance inspector or technical manager determine how to divide oversight effort. Each time a method is selected in Eclipse, the three quantities are determined for that method, and corresponding sounds are simultaneously presented in stereo, one from the left speaker, one from the right speaker, and one in the center of the stereo image. Two of the sounds, rain from the left speaker and a water stream from the right, are drawn from nature. The center sound is that of a cello. Each quantity is proportional to the amplitude of its mapped sound. A parallel, visual mapping showed each quantity as a numerical text-based value. The audio and visual mappings were used alone and together. Subjects were asked a number of questions comparing the values seen or heard among different methods within a software program. They were also asked to provide some conclusions, such as which method was seen as the most important based on the mappings seen or heard. Finally, they were asked their preferences.

The number of correct answers was not only equal but also had one-to-one correlation, demonstrating that the quantifications can easily be discerned, even in a three-way simultaneous presentation. The asked-for conclusions were all answered using either the visual or audio mapping, but response time was longer for the audio mapping by five to eleven seconds. The researchers feel that their study presents strong evidence that non-speech audio can be equally as effective

as visual approaches, particularly "if used in the right context and with the right program information data."

One participant in the Hussein study stated such a context, that in which one can scan the methods visually, leaving the details to audio. Thus, they would not have to constantly shift visual context from the scan to the details of each entity. As shall be seen, several manifestations of this idea are explored in the research at hand, in which off-screen details are presented and entities' characteristics are explored as being one set of the "right program information data."

The users in Hussein's study preferred visual presentation over audio presentation, some questioning the practicality of audio for the chosen purpose, but they also expressed interest in further exploring the audio approach. Much as Stefik postulates a non-trivial learning curve for speech-based audio, Hussein's study raises the possibility that there exists a comfort curve for non-speech audio which, if not reached, may negatively impact its adoption. Hussein points to the newness of the use of non-trivial, non-speech audio by his subjects for their lack of comfort with it, a symptom of cultural bias toward the visual for analytical purposes. However, it is also possible that the auditory experience is *not realistic enough*, in spite of the use of natural sounds and the cello. A sound from the "left speaker" is not as realistic in the listener's auditory space as a sound placed in the left side of the stereo image and assigned a simulated distance from the listener through the use of local and global reverberation, let alone binaural techniques. The rain, water stream, and cello exist in separate auditory spaces. Moreover, simple use of higher amplitude to represent a greater numerical value is not realistic. When an instrument, such as a trumpet, plays louder, its timbral spectrum also changes, and if one wants a trumpet to be really loud, one uses two or three of them, introducing slight variations in intonation and vibrato. Recall that the Peep Network Auralizer increased the number of chirping birds, not only their overall amplitude [55].

3.9 Summary

This chapter has provided a review of literature regarding sonification, especially as employed in program comprehension. That review included a discussion of sonification and audio display in general, design guidelines, and pertinent material from semiotics and cognitive psychology. A section of the chapter was dedicated to audio engines and tools for digital realization of sophisticated sounds. Finally, sonification in program comprehension was discussed.

Chapter 4

Listening to Software Structure

This chapter describes an approach to the use of non-speech sound to assist the comprehension of static program structure. The approach consists of the selection of a set of questions that can be addressed via non-speech sound, followed by a solution containing:

- a reference sound mapping
- a tool
- generalized guidelines for sound mappings

The chapter is divided into two main sections. In Section 4.1, *The Concept*, the approach and its components are introduced, and that which is new and unique is pointed out. Section 4.2 presents feedback that occurred during formulation of the sound mapping, resulting in decisions which informed the final mapping. Throughout the chapter, the term *developer* refers to the person involved in program comprehension who would be using the tool. The term *entity* refers to a package, class, or other specific software entity, and *entity type* refers to the idea of package, class, etc.

4.1 The Concept

As a program is understood, its entities are identified, differentiated, classified, and related to one another by the developer. The chosen sound universe and mapping rules must support that process, and a set of tools must support actions by the developer that facilitate that process.

Accordingly, the solution incorporates sound patterns representing software entities and their static characteristics into an integrated development environment (IDE). The developer performing comprehension of a sizable program interacts with the IDE to listen to source-code entities, their characteristics, and the entities that relate to them. The developer can make determinations of a structural and functional nature, even though entities may not be the focus of visual attention or even in within the visual field.

The solution is realized using synthesized and captured sounds applied to Java programs. It is intended to supplement visual features of the IDE for sighted developers.

4.1.1 Low-Level Java Structure

The low-level structure of a Java program indicates the organization of and relationships among the program's source packages, classes, interfaces, and methods. It is static in that it is written by a developer either directly or via some code-generation process prior to runtime. Source generated dynamically during runtime using the Reflection classes [106] or their equivalents are not considered, though the sound mappings and sound formulation principles introduced in this study would appear to apply to it.

Previous sound-related studies, summarized in Section 3.8, have focused on hearing the dynamic, runtime aspects of program comprehension. That is, they have concentrated on listening to the program's execution as it unfolds over time. Either the algorithm is heard as it unfolds or the state of the program can be

heard while it is paused at a debugger breakpoint. As such, previous sonifications reveal information such as what method is being currently called or what the current nesting level would be. The sonification concept presented herein reveals structural and semantic information as designed into the program, such as all the classes and methods referenced by or referencing a selected method, whether a method is static, and whether a method's function is data access.

The Java entity types sonified in this study, and program comprehension questions whose answers are sonified, are described in the following list.

- *Package.* A package is a container for classes and interfaces. In a sizable software project, the entities collected into a package are usually related by subject, architectural function, or both. A subject is something in the problem or design domain. A package called *editors* would be expected to contain entities that implement editors. A package called *dataAccess* would be expected to contain code at the data-access level of a layered architecture. Sonifiable program comprehension questions concerning packages are:
  - Which package is this?
  - What classes, interfaces, and methods belong to this package?
  - What code external to this package makes use of code within it?
  - Does this package have a particular architectural function? E.g., does it constitute a data access layer?
- *Class.* A class is a collection of attributes, such as variables, and functions known as methods. These attributes and methods are said to be encapsulated by the class, in that their visibility outside the class can be controlled. A class may be active or passive. Active classes contain methods that implement significant logic. Passive classes only provide access to encapsulated run-time data. Sonifiable program comprehension questions concerning classes are:

  - Which class is this?
  - Does this class implement any interfaces?
  - If so, how many and which interfaces does this class implement?
  - What methods belong to this class?
  - What other packages, classes, and methods does this class reference?
  - What other packages, classes, and methods reference this class?
  - Does this class have a particular architectural function? E.g., is it strictly a data access layer class?
  - Is this class active or passive?
  - Approximately how many methods does this class contain?

- *Interface.* An interface is a blueprint or template for a class. It specifies the signatures of externally-visible methods that a class must implement in order to conform to it. Sonifiable program comprehension questions concerning interfaces are:
  - Which interface is this?
  - What methods does this interface implement?
  - Which classes implement this interface?

- *Method.* A method is a function uniquely identified by its signature: its name, return type, and formal parameters. Some methods provide significant program logic, while others only provide access to run-time data. The latter are known as *accessor* methods. An accessor method is usually named *get*, *put*, or some variation thereof such as *getEmployee*. It may contain logic that adjusts the data's form, but it does not provide program control logic. A method may be a constructor, which has the same name as its containing class and which is called when an instance of the class is instantiated. A

method may be a finalizer, which cleans up memory references when an instance is no longer to be used. A static method belongs to the class itself rather than an instance of the class. Sonifiable program comprehension questions concerning methods are:

- Which method is this?
- Which package and class contain this method?
- Is this method a constructor? A finalizer?
- Is this method static?
- Is this method an accessor method?
- Does this method strictly perform data access?

4.1.2 Sound Universe

A rich, adaptable set of sounds and sound patterns is needed to sonify the Java entities and their characteristics to meet the challenges expressed in Section 3.6. The mappings of sounds and sound patterns to software entities must be coherent and readily learned. The developer should be able to categorize each entity according to its structural and functional characteristics yet also be able to differentiate entities belonging to the same category. The developer should also be able to garner at least approximate counts and sizes when presented aurally. Finally, the developer should be able to determine associations among entities. Thus, the four outcomes of the sonification of each entity are classification, identification, counting, and membership.

The sound universe is partitioned according to a three-layer distance metaphor mapped to the hierarchy of entities. While a three-partition design is driven by the number of software levels (interfaces being similar enough to classes to be placed in the same level), it is also well within the number of chunks that can be held in short-term memory [110] should the listener have to mentally contrast them.

Packages are fundamentally represented by outer-space kinds of sounds, classes are fundamentally represented by air-like or wind-like sounds, interfaces are birds in the air, and methods are represented by any sound occurring at the surface of the Earth. The sounds for packages and classes let the listener know an entity is a package or class, respectively, while superimposed musical motifs provide identification of the specific package or class. Superimposition is not employed for interfaces (birds) or at the Earthy level of methods. Methods can be represented via a wide variety of natural, mechanical, and musical sounds. If one hears such a sound pattern without an accompanying spacey or air-like sound, one has heard a method. An exception is any bird song, which represents an interface, birds being associated with the atmosphere. The metaphor is shown in Figure 4.1.

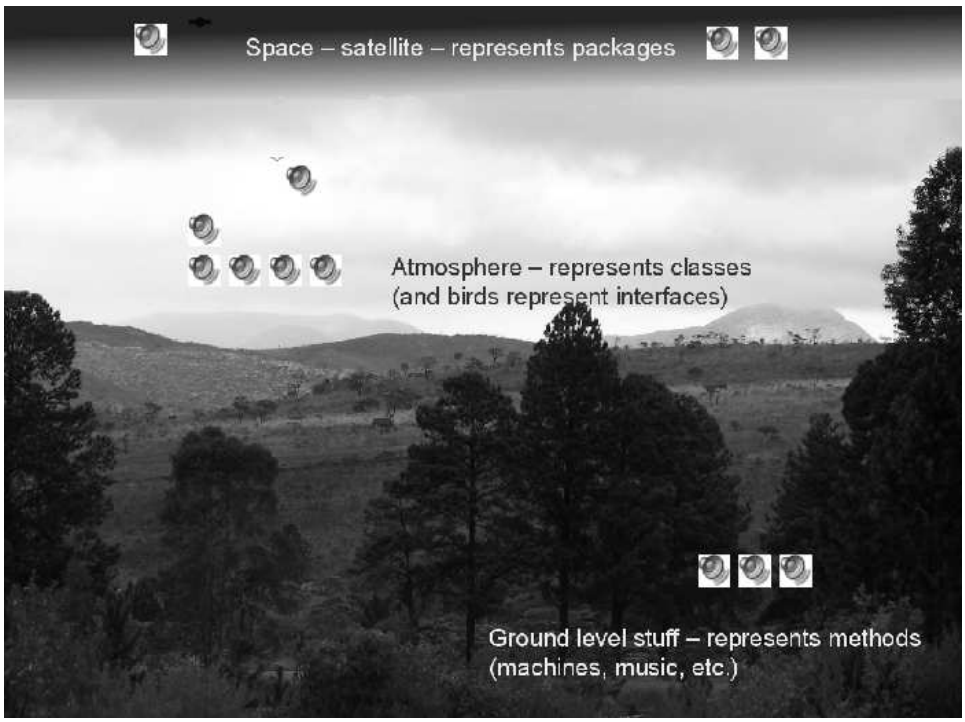

Figure 4.1: Metaphor for software entities

A method's timbre or pattern can indicate something about that method's role. For example, a constructor is always represented as set of hammer strokes on wood. If a class has overloaded constructors, each is represented by a different number of strokes. A finalizer is represented as the sound of a machine turning

off. Every finalizer sounds exactly the same. Every constructor sounds the same but may have a different number of hammer strokes. Accessor methods are always bell-like sounds. Data readers and writers are strictly upward or downward musical patterns, implying that the data are being written ("put up to") or read ("taken down from").

A method can also be augmented through the use of a modifier, a sound occurring immediately before or after the identifying pattern. For example, a static method is immediately preceded by an anvil stroke.

Class size is represented by drums playing at different intensities. A consistent drum sound is used to represent each of three ranges of sizes, as shown in Table 4.1. Duration is measured without reverberation. The three class sizes are described and heard starting at 11:08 in the training audio stream.

no. of methods	**representation**	**Duration (Sec.)**
0	single, quiet bass drum stroke	0.2
1 to 10	five moderate bass drum strokes	0.5
11 or more	many loud bass drum strokes	1.2

Table 4.1: Mapping bass drum to class size

It is valuable to know whether an entity is within or outside the project of interest. Entities outside the project are entities found in external libraries such as *java.io*. Outside entities may be encountered as classes and methods referenced by an entity within the project, classes and methods referencing an entity within the project, or classes and interfaces inherited by or implemented by an entity of interest. The distinction between entities within or outside the project is realized through differences in audio distance, where outside entities are heard as far and, secondarily, to the left or right. The distinction is characterized in Table 4.2.

The sound universe employs auditory icons and earcons, as described in Chapter 3. The invariant turning-off sound that represents a finalizer is an auditory icon. Most entities, represented by earcons, are more like the static data writer method, for whom we hear an anvil followed by a unique, upward musical motif.

Where	**Audio Distance**	**Realization**
within the project	close	normal volume and reverberation, placement may be centered or off-axis
outside the project	far	reduced volume, increased reverberation, and off-axis placement

Table 4.2: Mapping entities within or outside the project by audio distance

The anvil provides categorization of the method as static, and the upward motif identifies the method's function as writing data.

4.1.3 Design Process

The sounds and sound patterns are designed to meet certain goals:

1. The sounds should be coherent.
2. The sound mappings should be readily learned.
3. Related sounds should possess commonality according to structural and functional characteristics of the represented entities.
4. The sounds should be differentiable from one another.
5. Sounds should be aesthetically pleasing (unless intentionally harsh for representational purposes).

The design process employed to obtain the sounds is summarized in Figure 4.2 and described below.

A concrete foreground sound is selected if there is an obvious mapping. For example, a file open method could map to a door opening sound. Concrete sounds are either recorded or selected from a library of available sound effects. If no obvious mapping occurs to the designer, a sound of a more abstract nature must be generated.

If semantics are to be indicated, an attempt is made to generate a sound pattern evocative of the semantics. For example, a factory method [52] could be

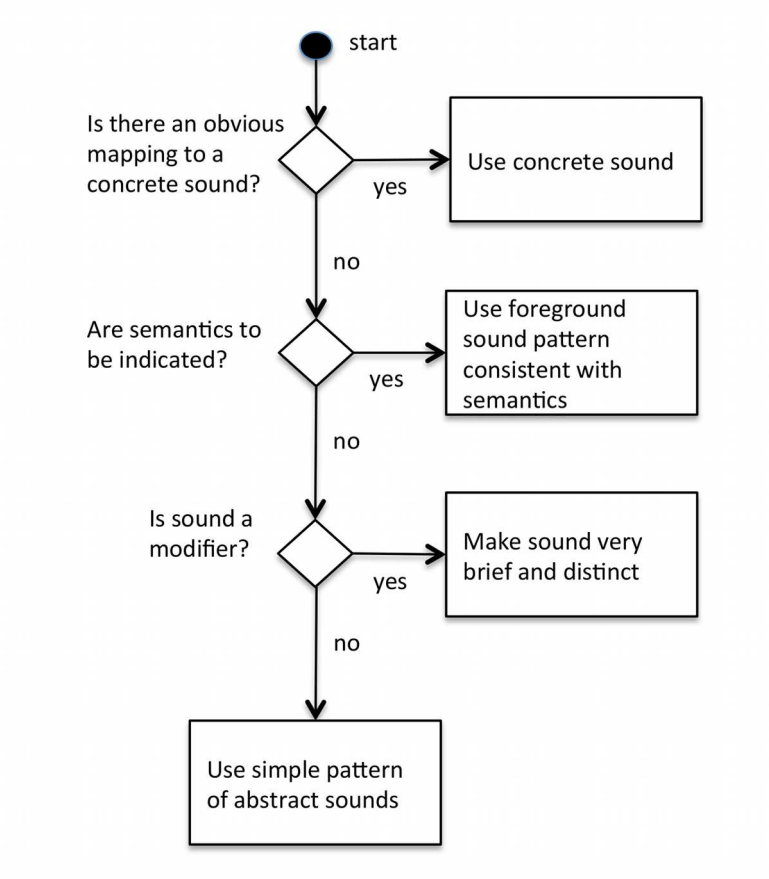

Figure 4.2: Design process

represented by a sequence of mechanistic sounds arranged in a staccato pattern to capture the workman-like, mechanistic quality of a factory. The pitches in the pattern do not correspond with diatonic musical notes so that the mechanistic quality rather than a melody is emphasized. A program's starting point, its main method, could be represented by a heralding pattern played by a simulated trumpet, evoking an entrance or welcoming.

The default type of foreground sound representing an entity is a simple musical pattern played by a simulated instrument or collection of instruments, most often a flute or clarinet for their pleasant qualities and ability to cut through any background sound. Initially during design, the foreground motifs are kept simple, being made more complex when further patterns are needed for subsequent entities. Timbres (flute, clarinet, etc.) are varied to help distinguish the sounds from

one another. The motives are generally kept under several seconds in length.

Modifiers are kept even shorter, a half second to a second in length. Modifiers shoud be attention-getting and quickly recognized. Hence the use of an anvil stroke for to indicate a static method: a single, accentuated sound employing a unique timbre. Bird calls are kept brief, as they can be used as modifiers indicating their inclusion in a class.

Any background sound should be transparent, that is, allow foreground sounds to be heard simultaneously. The wind-like sound in the space-air-earth metaphor meets this criterion. The space-air-earth metaphor itself is adopted as an aid to recall what kind of entity is being represented.

Sound representations have been subjected to iterative design. The sounds were played for volunteers who were queried as to concrete sounds' realism and other sounds' ability to be recognized, consistency with related sounds, and overall discernability. Suggestions were offered, and the sounds' designs were refined.

4.1.4 Reference Sound Realization

The mappings of sounds to entity types, employed in the reference realization, are summarized in Table 4.3. The Examples column contains the timings of representative examples in the training audio stream. The variety of method characteristics is summarized in Table 4.4, following. The set of characteristics employed is clearly not exhaustive, but it is large enough for the purposes of this study.

Packages

A package is represented using an underlying sound reminiscent of outer space, as it is at the highest level in the distance metaphor of the hierarchy of packages, classes, and methods. As there is, in reality, no sound in outer space, a satellite, an object associated with outer space is substituted. The underlying sound of a package is similar to those we hear in the media for satellite transmissions. This underlying sound is invariant among all packages.

Entity Type	Sound-Space Metaphor	Differentiation from Others	Identification	Modifiers and Indicators	Examples (Training Stream)
package	outer space	underlying, invariant satellite-like sound	pattern of overlaid beeps		3:06, 3:17
interface	atmosphere	always a bird call	unique bird call		5:29, implemented by class 5:50
class	atmosphere	underlying, invariant wind-like sound	instrument-like musical pattern of one to seven notes	*Size* - bass drum-like sound after the class sound itself, indicating approx. number of methods in the class	11:22 thru 12:02
method	earth	any sound associated with our ground-level experience vs. space or atmosphere	unique pattern from a wide variety of potential sounds, such as machine-like sounds, a shopping cart (for an add-to-cart method), or instrument-like musical sounds.	see Table 4.4	6:25, 13:24

Table 4.3: Mappings of sounds to entity types

Characteristic	Differentiation from Others	Identification	Examples (Training Stream)
Standard Method Types			
constructor	Sound of hammering wood	Overloaded constructors have different numbers of hammer strokes	8:08
finalizer	Sounds like a machine turning off	None; all finalizer sound the same	8:33
static	Identifying sound preceded by anvil	N/A	9:30
Method's Functionality or Architectural Role			
accessor (get, put)	Simple bell-like sounds	Unique pitch or bell-like pattern	6:36, 7:10.
writer	Strict upward pattern of notes	Unique pattern	10:07, 10:32
reader	Strict downward pattern of notes	Unique pattern	9:55
reader-writer	Strict upward followed by strict downward pattern	Unique pattern	none
factory method	Musical phrase with machine-like timbre	unique phrase	none
Class Characteristics			
this class	Class with single cello tone in foreground	N/A	13:19

Table 4.4: Mappings of sounds to method and class characteristics

Above the underlying sound, in keeping with the satellite metaphor, a unique pattern of beeps identifies the particular package. Differentiation among packages is achieved by varying the number of beeps, their frequencies, and their durations. This does not provide the means to differentiate a huge number of packages from one another, but the number if packages in software projects is much less than number of classes and methods, so a design trade-off has been made in favor of keeping with the metaphor.

Classes

A class has an underlying sound of wind, keeping with an atmospheric metaphor for the second level of the distance hierarchy. As with packages, the underlying sound is invariant among all classes. Above the underlying sound is superimposed a musical pattern by a (digitally simulated) wind instrument. The musical pattern or phrase identifies the particular class. Should the number of classes exceed that which can be reasonably differentiated by wind instruments, other instruments such as a (digitally simulated) lyre can be used. The instruments are digitally simulated so that the patterns or phrases can be generated rather than having to be pre-recorded.

Interfaces

Given that a wind-like sound underlies classes, and given that it is desired to reveal which interfaces are implemented by a class, interfaces are all represented as bird calls, and different bird calls differentiate different interfaces. As these bird calls do not have a common underlying sound, they are free to be heard standalone when listening to the interface itself yet also be heard superimposed on an underlying class sound when listening to the interfaces that the class implements. In the latter case, the bird call follows the class identifying phrase, both of which are superimposed on the underlying class sound.

Class size

Drum sounds are used to depict class size in terms of number of methods. In-

tensity and number of repetitions vary together from a single, quiet drum stroke, representing a class with no methods, to a loud, repetitious series of drum strokes, representing a large class. Hypothetically, as classes may vary from zero to hundreds of methods, a continuous spectrum of gradations can be employed, the bounds configurable by the developer using the tool. For purposes of the present study, three discrete gradations are employed: one for a class that has no methods, one for a class having one to ten methods, and one for a class having more than ten methods. In an exploratory capacity, the discrete set would still offer the developer a heuristic for which classes to focus on.

Methods

The third distance layer is the surface of the Earth. Since that is the realm of our common experience, any sound not of outer space or the atmosphere can represent a method. For this reason, there is no need for an underlying sound; a simple auditory icon or a standalone musical phrase can represent a method. This also affords the opportunity, if desired, to superimpose a classes' methods above it.

Methods in a software project may number into the thousands, requiring a wide representational variety. Standalone musical patterns, drawn from a wide variety of digitally-generated instruments, can be used. The timbres, phrase structure, and articulation can help differentiate methods from one another. Alternately, auditory icons composed of "real-world" sounds can be used, especially when some important semantic aspect of the method is to be expressed. For example, the sound of a door closing is used to represent a *close()* method.

A class constructor has such an analog: the sound of hammering wood in an outdoor environment (i.e., someone constructing a house). Constructors may be overloaded; multiple constructors within a class are differentiated through different numbers of hammer strokes. On the other hand, two constructors, one in each of two classes, may have exactly the same auditory hammering icon, as hearing the

difference in the classes themselves is sufficient to determine that they are different methods.

It would be desirable to hear the number and type of a method's formal parameters, especially in the case of overloaded constructors. The notion was omitted from the sonification scheme, however, as it can be considered a detail of the method, and adding additional sounds to the method representation would make it more complex and increase cognitive load. Surely the number and type of formal parameters lives at the interface to other methods, but it is questionable whether this information is important while working in an exploratory mode.

A class finalizer is always represented by a simple auditory icon depicting a machine (actually a vacuum cleaner) turning off. Given that there is only one finalizer per class, named *finalize()*, there is no need to differentiate overloaded finalizers as with constructors.

Accessor methods, those which simply "get" and "put" encapsulated memory variables, are mapped to simple, bell-like sounds, the bell sounding a small number of times. The unique pitch and timbre of the bell differentiate the methods.

Methods dedicated to reading and writing data are common. Readers are characterized by an upward musical pattern, as if "putting data up" to its destination. Writers are characterized by a downward pattern, as if "pulling data down." There may be many readers and writers in a project, so they are differentiated by different actual patterns, each of which retains the upward or downward aspect. Each of the two patterns shown in Figure 4.3 would be writers.

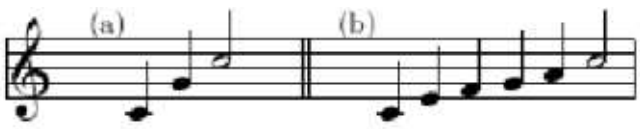

Figure 4.3: Two writers as strictly upward patterns

The short pattern to the left can be used to represent a writer that performs a single write, while the one on the right represents a writer that does many writes

or calls subordinate writers. That determination is up to the discretion of the human or machine generator of the mapping. Other, similar informal, possibly program-specific semantics may be so added.

The present study just touches upon the idea of mapping sounds to architectural design patterns. A simple design pattern is the factory method, which is represented by a quasi-musical phrase having a machine-like, highly enharmonic timbre.

A method may be static, that is, belong to a class itself rather than an instance of the class. Because the *static* keyword in a declaration appears before the name of the method, so the auditory static modifier, an anvil stroke, appears immediately before the method identifier. The anvil stroke is brief (about 1/4 second in duration) to emphasize its use as a modifier.

Internal versus external entities

A sonified entity may be external to the project(s) shown in the Project Explorer. This can occur when listening to referenced or referenced-by entities. For example, the entity selected in the Package Explorer may instantiate classes and call methods in the *java.io* package in the Java API. The referenced package, classes, and methods will be heard as being noticeably more distant from the listener. Sonic distance is implemented through decreased volume and increased reverberation, and it is reinforced by off-center placement of the sound within the stereo image. Figure 4.4 depicts close versus distant entities in an auditory space.

Generalization: mapping to auditory icons and earcons

To summarize the sounds and patterns presented above, individual software entities are mapped to audio constructs in a continuum from simple auditory icons through complex earcons.

The simplest entity mapping is to a single auditory icon. All *finalize()* methods are mapped to a single auditory icon. Its character indicates that the entity

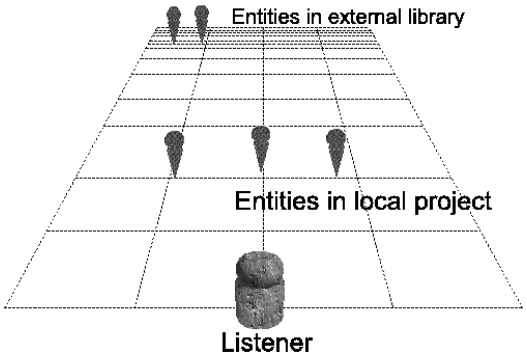

Figure 4.4: Close vs. distant entities

is a method, as the sound is not of outer space or atmospheric, and the specific, pre-determined sound indicates that it is a finalizer. No differentiation from other finalizers is present or necessary. Constructors are similarly mapped. When constructors are overloaded within a class, their mappings are to a class of similar auditory icons that can then be seen as primitive earcons in that they differentiate the individual entities.

Package and class mappings combine into one construct an underlying auditory icon with a unique, primitive earcon drawn from a set of sound patterns that differentiate the specific entities. A method is mapped to an earcons in that it may carry semantic information such as whether the method is static or that it is a reader. Finally, class size is mapped to the truest earcon, representing a continuum according to intensity and drum-roll duration.

Some of the sounds are deterministic, such as the hammering for a constructor. With others, such as the bell sound for a *get()* method, timbre is deterministic while pitch is arbitrarily chosen by the human or automated mapping generator. The upward motion of a writer is predetermined, but everything else about its sound mapping to a musical phrase is arbitrary.

Listening to multiple entities

Presentation of a package, all its classes, and all its methods is accomplished by presenting each entity in sequence. Thus, we linearize a tree structure into

a sequential succession of sound constructs. One construct is separated from its predecessor via silences measured in seconds. It is envisaged that the developer can set the number of seconds between sound constructs as a configuration parameter. Listening to the parents of a method would entail listening to the method, followed by its parent class and finally the parent package. Sequential presentation may slow comprehension due to its length, but that effect may be counterbalanced by avoiding search.

Using the Sound Mapping

The sound mapping has been designed with a mature version of the tool in mind. A developer might set what he wants to listen to in the Sonification View, then hover over various entities in the Package Explorer or editor. The developer might perform such exploration in sequence, momentarily hovering over items of interest, or more at random, or possibly according to some heuristic. Upon hovering on an entity, the desired characteristics of that entity would be heard.

The sound mapping is designed to address exploratory questions posed by the listener with respect to entities as emerging beacons. Presume the developer has no prior knowledge of two classes, *Response* and *Responses*. Without having to expand each class in the Project Explorer, the developer visits them in turn, having chosen to hear classes and their child methods. The set of methods within *Response* is heard in sequence. The developer notes that there exists a constructor (mapped to the wood hammering pattern), some general methods (mapped to general standalone sounds) and a variety of *get()* and *put()* methods. Then the developer listens to the *Responses* methods in sequence. Here there are mappings to a lot of static reader and writer methods (each an anvil followed by an upward or downward musical phrase). The developer may well postulate that *Response* is a domain-related object that holds some kind of response, possibly a textual response to a developer, and *Responses* is a helper class that retrieves and stores

serialized *Response* objects, possibly in a data store known as "Responses." The developer would eventually test these hypotheses, but may postpone testing to continue exploring other entities.

The developer may discern other information about a class by hearing the serialization of its methods. Simply by hearing a class followed by only accessor methods, the developer can discern that the class itself is passive, performing no logic other than that for managing its encapsulated variable. If the developer is trying to find algorithmic logic, this can quickly be determined not to be a class of interest.

A method of interest may call or be called by other methods external to the software project under consideration. Listening to these, the developer can gain a sense of the coupling points between the project of interest and external code. For example, the developer can determine that a local class is highly dependent on methods in the library package *java.io*. Its classes and methods are heard as distant and off-center. Note again that the developer may never have had to expand packages or classes in the Package Explorer, and the developer never has to visually visit the external package to learn that certain methods are data readers and writers.

Shneiderman's concept of gaining knowledge via survey, zoom, filter, and details [142] is fulfilled by combining visual information and navigation with the audio constructs and sequences. Navigation is performed in a traditional, visually-oriented way, via mouse-over. Listening may be done at a package level; when the developer finds a package of interest, the developer may zoom by expanding the package and listening to the classes in turn. The developer filters by choosing what characteristics he would like to listen to. Finally, summary-level details such as whether a method is static is available at the audio level, with further details available through traditional, targeted code reading.

Applicability to Other Languages

The sound mapping has been constructed for the Java language, but it should be readily adaptable to similar languages of a combined object-oriented, procedural nature. Examples of such languages are C# [108], C++ [160], Ada95 [163], and Python [98]. Each has the equivalent of classes, constructors, and inheritance as well as the equivalent of Java namespaces. Multiple inheritance in C++ can be accommodated by the existing mapping. The C# language includes *properties*, which incorporate accessors and mutators as language concepts, making them appear as variables to referencing code. This increases the usefulness of the existing sound mappings for accessors and mutators, as those sounds can help differentiate accessors and mutators from other kinds of variables during code reading without having to open and inspect them. New modifiers (such as the anvil sound for "static") can be devised for language features such as C# *delegates*.

4.2 Feedback Results

Informal feedback was solicited during formulation of the mapping concept. The results were incorporated into the reference mapping and the prototype tool. Those consulted were experienced in either the programming discipline or the music discipline. The former were posed questions such as, "does this sound effectively represent a class, and if not, why not?" The latter were posed questions such as, "does this construct appear to represent one thing or multiple things?" Concepts were incorporated or rejected based on the informal feedback, interviews and brainstorming. The feedback is presented here in essentially the form of an experience story [58]. Specifically, feedback impacted the presentation of class size representation, the parallel versus sequential nature of entity presentation, and connection of referenced entities via an "arrow" sound. Sizes were best communicated through differences in intensity and temporal parameters rather than

pitch. Sequential presentation of parent-child entities such as a class and its methods was chosen over parallel presentation. An arrow sound was dropped from the sonification scheme.

4.2.1 Class Size Presentation

A concern identified early on was how quantitative information of a non-temporal nature, namely the size of a class expressed as its number of its methods, may best be discerned. Initially, mappings of pitch to number of methods were devised, mapping pitch to size, with higher pitch meaning larger size. Two mappings were considered: a linear mapping of named pitch (as opposed to frequency) to size, and an exponential mapping in which a pitch class (such as C) at each octave is one power of ten higher than the previous. Refer to Figure 4.5. Under either mapping, either a single pitch was heard representing the class size, or a scheme was employed in which the lowest pitch was heard, then a glissando (slide) would occur up to the pitch representing size to try to provide a relative rather than absolute frame of pitch reference.

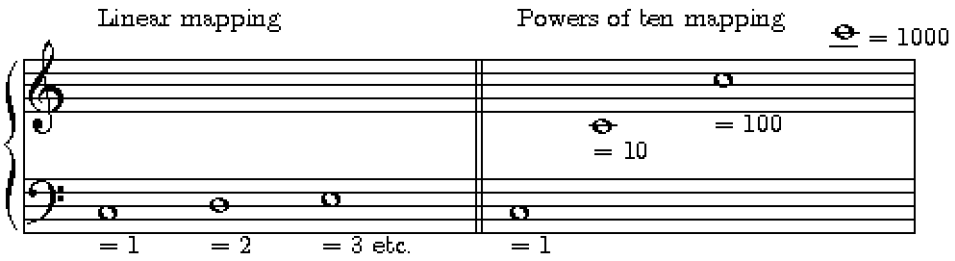

Figure 4.5: Pitch mappings to size

The persons to whom these approaches were informally presented, several very musical, others less so, all performed poorly mapping pitch to size. Thus, these approaches were disfavored as not being intuitive and were not further pursued. A hypothesis is that increasing pitch is antithetical to the knowledge that the

data value is a size, which implies a larger, denser, or louder sound. Further, in the glissando variation, it takes time to get from the reference pitch to the target pitch, introducing an explicit temporal element. A hypothesis is that the introduction of an obvious temporal element destroys the metaphor of a single, time-independent data value. Finally, the approach does not present a zero value clearly distinguishable from other values.

Mapping pitch to size was abandoned, and the current approach was adopted in which increasing number of sub-events (drum strokes) is reinforced by intensity to fulfill the metaphor of increasing size. Also, size has been divided into ranges, as it is sufficient to know the approximate number of methods while working in an exploratory mode. As will be seen, this mapping was successfully realized in the validation study.

There are three ranges, the sound mapping for each depicted in Figure 4.6. A single, quite drum stroke represents a value of zero: the class contains no methods. A brief drum outburst at medium intensity means there are from one to ten methods. A longer drum outburst means there are more than ten methods. This

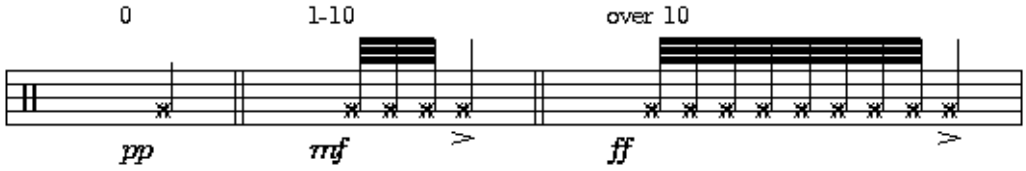

Figure 4.6: Drum mappings to size in number of methods

division into ranges is coarse. Future work will be required to determine just how many ranges of values and gradations of drum rolls can be usefully recognized.

4.2.2 Entity Presentation

Under the mature concept, related entities such as those in parent-child relationships are presented sequentially. Originally, the intent was to superimpose parent-child entities. A class would be represented by a wind-like sound, con-

structed via granular synthesis. Two or three such wind-like components might appear together in a kind of chord. The center frequencies, densities, and periodic swells in the sounds would uniquely identify the class. The class sound would persist while the developer explored methods within the class, shifting to a different class sound when the developer moved the pointing device into the browser area covered by another class. The package sound, as currently constituted, would be heard occasionally to remind the developer what package the class belonged to, and method sounds as currently constituted would be heard as methods were pointed to. The early concept is presented musically in Figure 4.7.

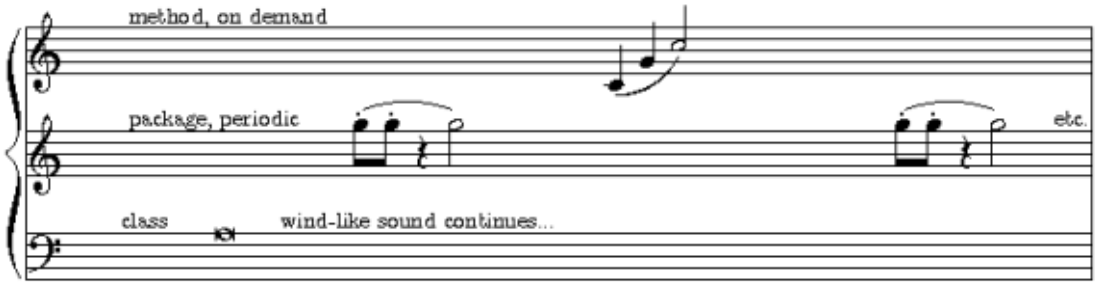

Figure 4.7: Early concept

Listeners informally presented with the early concept found it difficult to perform differentiation of classes based purely on the differences in wind-like sounds. They also felt that it was not necessary to be reminded of the class or package on an ongoing basis, favoring an on-demand basis. Also, the periodically repeating nature of the package sound was antithetical to the sense of user control and on-demand feedback. Finally, it was unclear whether modifiers added to the sound baseline would apply to the class or method being examined. It became apparent that it would be more intuitive to present an entity and its modifiers as a single cluster of sound in time.

Feedback during early developmental stages also indicated that an arrow sound as described in Chapter 2 had been indeed unnecessary. An arrow sound was designed that would be inserted between related entities. It would move upward

in pitch if the second entity was in a higher hierarchical position than the first (e.g., its parent), downward if lower (e.g., its child). It was to be used mainly for called-by and called relationships. It was deemed unnecessary because the listener selects what to listen to, therefore knowing that called-by or called entities are about to be heard.

4.3 Summary

This chapter has presented a novel concept for listening to software structure at the level of Java entities: packages, classes, interfaces, and methods. Each entity is represented by a sound or a time-bounded set of sounds, some of which overlap and some in sequence as determined by the entity's type. An earth-air-space metaphor for the sounds of the entity types serves as a learning and memory aid. Related entities, especially those in parent-child relationships, can be heard in sequence, separated by brief silences.

Chapter 5 describes a prototype tool that realizes the concept.

Chapter 5

Prototype Tool Implementation

This chapter describes the tool that was developed and used experimentally. Section 5.1 introduces the concept. Section 5.2 describes the tool. Section 5.3 briefly describes how sounds were constructed and provides attribution. The final section summarizes the tool and reference sound mapping.

5.1 Introduction

A prototype tool was designed and built to demonstrate concept viability, facilitate the establishment of sound mappings and realization guidelines, and support a human-subjects study. As it became clear that the sound mappings and guidelines would become the dominant issue to address, tool development was bounded to support the mapping, with advanced feature implementation deferred for future study. A working prototype used is described in this section.

The tool consists of an Eclipse plug-in and a Csound audio back end. The developer using the tool can select items in the software project being worked on in Eclipse, listen to them, and listen to entities related to them. The plug-in sends Csound score statements to Csound, which in turn produces the sound.

The tool departs in several key respects from those described in Chapter 3:

- the tool extends static IDE views of the software with sound. Specifically, it

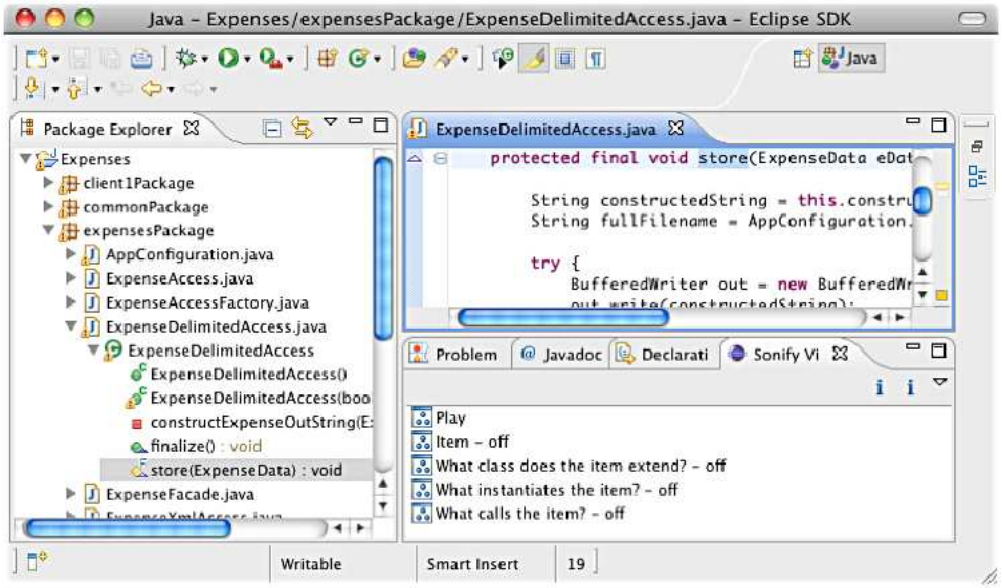

Figure 5.1: Eclipse user interface for sonification

extends the Eclipse Package Explorer. It can be easily adapted to support item selection within the Java source code editor. Prior tools have focused on adding sound to run-time elements such as the debugger.

- The tool focuses on entities at the low-level architectural level: packages, classes, interfaces, and methods. Prior tools have operated at the statement level.

- The use of Csound as a back end provides ability to achieve the desired rich, generalized sound universe, including simultaneity, localization, and sophisticated sounds which are aesthetically pleasing.

- On-the-fly generation of Csound score statements offers the ability to construct sounds in response to the developer's navigation and selections.

Use

Figure 5.1 shows the Eclipse user interface enhanced by a *sonification view* provided by the sonification tool. Listening to entities requires use of the Sonification View, to the bottom right in the figure, and the Package Explorer, at the left. The

Sonification View allows the developer to choose what will be sonified in response to the selection of an entity. The Package Explorer is the venue for selecting an entity.

The Sonification View in the prototype tool provides the following listening options:

- *Item* - the entity selected in the Package Explorer.
- *What class does the item extend?* - entity inherited by the class selected in the Package Explorer. Selected entity must be a class.
- *What instantiates the item?* - entities that instantiate the class selected in the Package Explorer. Selected entity must be a class.
- *What calls the item* - Entities that call the method selected in the Package Explorer. Selected entity must be a method.

The Sonification View in the prototype tool does not contain the following listening options intended for future versions:

- *Item's Parents* - the containing package, if the selected entity is a class. The containing package and class, if the selected entity is a method.
- *Item's Child Classes* - classes belonging to the selected entity, if the selected entity is a package.
- *Item's Child Classes and Methods* - classes, interfaces, and methods belonging to the selected entity, if the selected entity is a package. Methods belonging to the entity, if the entity is a class.
- *Referenced Packages* - any packages referenced by the selected entity.
- *Referenced Methods* - packages, classes, and methods referenced by the selected entity. Thus, the prototype allows the developer to listen to entities

that reference the selected entity, some of which may be off-screen, but not entities referenced by the selected entity.

Double-clicking the *Play* button in the Sonify View actually plays the sound patterns.

The reader may consider a scenario in which the developer selects, in either order, *What calls this item?* in the Sonification View and the *store(ExpenseData)* method highlighted in the Package Explorer in Figure 5.1. Upon subsequently clicking *Play*, the developer will hear, in sequence, the sounds for the *expensesPackage* package, the *ExpenseFacade* class, and the *store* method within that class. The package will be heard as the characteristic outer-space satellite-like sound, overlaid with its identifying pattern. The class will be heard as the characteristic wind-like sound, overlaid with its identifying pattern. The method will be heard as an upward musical pattern, reflecting its role as a data writer. The entities will be separated in time by brief but discernable silences.

5.2 The Tool

This section discusses the design of the prototype tool. The tool consists of an Eclipse plugin, a TCP/IP socket interface [62] from Eclipse to Csound, and configuration of Csound orchestra and initialization-time score files.[1] The tool's architecture is shown in Figure 5.2.

To produce sounds in response to a developer action, CScore statements are generated and sent from Eclipse to Csound via the TCP/IP socket interface, whose sending socket is created and controlled within the Eclipse plugin, and whose receiving socket is created and controlled Socketreader, a Java program. Socketreader, in turn, serves as an input stream to Csound. Score statements are interpreted upon receipt according to the instrument definitions in the orchestra file

[1]The tool has been implemented using Eclipse 3.3.1.1 and above, Java 6, and Csound 5 under Linux and Mac OS-X.

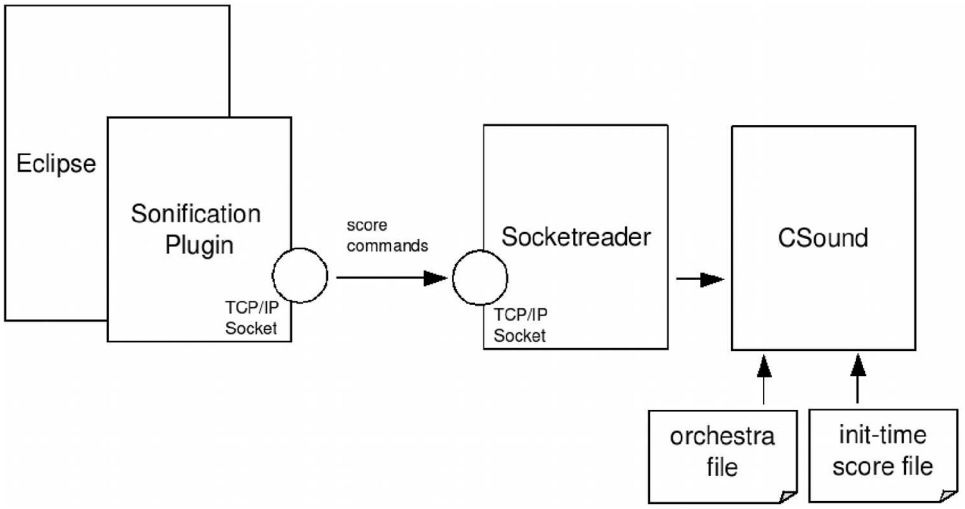

Figure 5.2: System Architecture

and the reverberation and other parameters made active by the initialization-time score file.

Eclipse Plugin

The Sonification Plugin has the following key components:

1. The Sonify View, with event handlers to process the developer selections and the *Play* button
2. Event handlers to process selections in the Package Explorer
3. Code which constructs score commands for Csound
4. The sending socket

When a button in the Sonify View is clicked, an event handler is activated. If the button controls what is played (*myself*, *my parent*, etc.), the event handler communicates this to the sound system. If the button is the *Play* button, the event handler invokes a method that causes the appropriate sound to be constructed and sent to Csound via the socket interface.

Event handlers are also invoked when items in the Package Explorer are clicked. Upon selection, an item becomes the "current" item, to be heard when the *Play*

button is subsequently clicked.

When *Play* is clicked, the plugin consults a stored list that maps each software entity to a set of score statements. Multiple score statements comprising a set are played at specified intervals. The following Java code associates a given entity, *commonPackage*, with a set of three sequentially-heard score statements:

```
e.add(new EntitySoundDescriptor(sonify.languageFeature.PACKAGE,
"commonPackage", "i10 0.0 5.58 45 1 16 7000\ni11 1.1 0.3 45 1
10000 891 1\ni112.9 0.3 45 1 10000 891 1\n"));
```

The string containing the concatenated score statements is sent through the socket interface. Each individual score statement is terminated by a line break (*backslash n*).

Socket Interface

The socket interface is straightforward. A Java method, *establishConnection()* in the Sonification Plugin, creates a new socket on a high-numbered port, establishing a connection with a listener socket in *Socketreader*. Immediately thereafter, *establishConnection()* reads the initialization score file and sends its contents through the connection, establishing Csound functions and starting the global reverberation instrument. Sets of score statements such as that above for *commonPackage* are sent in real time, the buffer being flushed after each send.

Csound Processing

The standard output of *Socketreader* is piped into the standard input of *Csound*. Upon receipt, each score statement is processed, and the output of Csound is directed to the computer's audio card.

When Csound is started, commonly-used waveforms such as an anvil sound are read in and preprocessed. A global reverberation instrument is started and directed to run for many hours, longer than the expected duration of any usage

of the system. This provides reverberation for participating instruments, giving the listener they are the impression of being in a consistent physical space. Its presence also keeps Csound running even when no real-time score statement is being processed.

Thereafter, score statements are processed in real time until the system is shut down. A typical collection of score statements is shown below.

```
i10 0.0 5.58 45 1 16 7000
i11 1.1 0.30 45 1 10000 891 1
i11 2.9 0.30 45 1 10000 891 1
```

The first line initiates instrument 10, defined in the Csound orchestra file, at time 0.0 (when received), for a duration of 5.58 seconds. It is heard at azimuth 45 degrees (directly in front of the listener) and elevation 1 (actually, elevation is not used in the stereo image), using function 16 (a prerecorded, preprocessed wind-like sound) at amplitude 7000. Overlaid on this wind-like background is instrument 11, first heard 1.1 seconds after the wind-like sound starts for a duration of 0.3 seconds, then heard 2.9 seconds after the wind-like sound starts, for a duration of 0.3 seconds. Instrument 11 is a computer-generated tone whose frequency in each case is 891 Hz, utilizing function 1, which describes the timbre of the sound. Thus, we hear the representation of a class whose characteristic is provided by instrument 10 and whose unique signature is provided by the two iterations of instrument 11.

5.3 Sound Construction

The audio realization as implemented in the prototype tool contain acoustically recorded and synthesized sounds. Some of the acoustically recorded sounds were captured using a portable, high-quality digital recorder. Others were obtained from free-use web sites. Some Csound instruments for producing synthesized

sound were written *ab initio*. Others were adapted from the *Csound Catalog* [23]. Wind-like sounds were achieved through granular synthesis, which had been demonstrated to effectively serve as a background for overlaid foreground sounds during sonification of program slices [11]. Flute-like sounds were produced using physically modeled instruments found in the Csound Catalog, and brass-like sounds were produced using additive synthesis.

5.4 Summary

This chapter has described a prototype tool that realizes the reference sound mapping described in Chapter 4. The tool consists of an Eclipse plugin, a Csound back end, and one-way communication software in between. The Eclipse plugin captures mouse events in the Eclipse Package Explorer and provides a custom view for selecting and playing sounds. The sounds it produces may be digitally generated or prerecorded. The sounds are further processed to achieve directionality and the sense of audio distance.

Chapter 6

Review of Evaluation Techniques

This chapter presents a review of possible evaluation approaches for software engineering problems of the type encountered in this thesis. The first section, *Introduction*, sets the scope for the subsequent section, *Review of Possible Approaches*, which discusses the advantages and disadvantages of a number of approaches to data collection and evaluation.

6.1 Introduction

Given that software engineering involves activity performed by people individually and in teams, empirical methods involving human subjects are an important and widely accepted means of evaluation. Quantitative methods have been employed in software engineering research since at least the 1980's, as described by Basili in his landmark paper [7]. Qualitative methods have been championed by Seaman, among others [137]. Quantitative and qualitative methods have been employed in the evaluation of software development methods and processes, cognitive aspects of software engineering, and tools. Execution of studies, whether quantitative, qualitative, or mixed, involves data collection followed by coding and analyzing the data.

Software engineering is a complex human activity. As such, it is not always pos-

sible to conduct controlled experiments. Often, observations must be made in the workplace, with the attendant challenges, to evaluate individual task performance and collaboration among peers. Researchers must also evaluate concepts and decision making strategies that exist only within the individual software practitioner's mind. For these reasons, empirical methods borrow techniques from psychology and anthropology. A general exposition of many of these techniques can be found in Cooper [36].

Clarke identifies three empirical research traditions: Conventional, Interpretivist, and Engineering [34]. The Conventional tradition is that which has emerged over a number of centuries, having been applied early on to the physical sciences through the work of researchers such as Galileo. Also known as the Positivist tradition, it is based on the assumption that there is an objective truth whose attainment can be approached through observation of objective reality [138]. Data collection is concerned with size, duration, and other measurable attributes of objects and processes. Analysis is often conducted using statistical methods. An exemplar of the Conventional tradition is the traditional quantitative experiment, which is discussed at length by Maxwell and Delaney [103]. Quasi-experimental designs and field experiments afford opportunities to make observations in real-world situations, but they offer control over fewer independent variables.

The Interpretivist tradition is based on the belief that humans create their own valid, socially-constructed truths, and that there is no single, objective reality [138]. Truths may differ across cultures or at different times within a single culture. Qualitative methods have emerged from the Interpretivist tradition, initially in the field of anthropology, to describe human understanding, communication, and understanding. Qualitative techniques are discussed thoroughly by Patton [120]. While a conclusion cannot be proved by qualitative methods, it can become accepted through a preponderance of evidence. Textual and pictorial data, rather than numerical data, are distilled to identify and describe common concepts, which

can be tested through further data collection and analysis. The coding scheme for qualitative data often arises out of the data itself. Patton points out a distinction between constructivism, whose focus is "the meaning-making activity of the individual mind," and constructionism, which focuses upon "the collective generation [and transmission] of meaning." This distinction can be blurred in terms of determining where ideas and feelings expressed via some data collection method originate - the collective certainly has impact upon the individual, individual ideas are adopted by the collective, and individual ideas can be critical of the collective. The distinction is clearer in operational terms, for example, examination of software processes shared by teams versus elicitation of thought processes of individual developers performing program comprehension tasks. Many of the data collection and analysis techniques described below can be applied in both constructivist and constructionist modalities. Examples of qualitative methods in software engineering studies include Von Mayrhauser and Vans, who employed qualitative methods to build a meta-model of the program comprehension process [176], and Das, Lutters, and Seaman, who uncovered characteristics of useful software documentation and the interaction between maintainers and documentation [40].

Software engineering efforts produce much quantitative data, such as the number of errors in a program or the duration of a project. Collection and analysis of quantitative data has been encouraged over the years through initiatives such as the Software Engineering Institute's Capability Maturity Model for Software, whose purpose is organizational software process improvement [164]. Software engineering efforts require much collaboration and thought, which can largely be expressed textually and pictorially. Because numerical, textual, and pictorial data are readily available or readily generated, quantitative and qualitative methods may coexist-exist in a single research project. The researcher may maintain a primarily positivist mind set, using qualitative analysis for explanatory or amplifying purposes and as motivation for further theory building [138].

Finally, research in the Engineering tradition proceeds by conceptualizing, building, testing, and demonstrating prototype technological artifacts and techniques [34]. An engineering approach is often used in the early stages of applied software development to mitigate project risk by verifying user interface concepts or the incorporation of a new technology. It often accompanies and complements requirements gathering and analysis. In software engineering research, it may complement or enable quantitative and qualitative methods by providing a vehicle for experimentation such as a tool. An example is the redesign of the SHriMP visualization system's user interface, in response to observations from a pilot study, to support the ensuing full study [155].

6.2 Review of Possible Approaches

Singer separates data collection techniques from evaluation techniques, indicating that a single data collection activity may be followed by distinct qualitative and quantitative evaluation [145]. For example, a questionnaire-based survey may contain quantitative questions on a Likert scale as well as space for free-form textual responses. Qualitative analysis of the text may help explain the motivation or mind set underlying the quantitative responses.

Major approaches for empirical studies of software practitioners are field studies, surveys, and formal experiments, the first two being primarily quantitative or qualitative, the latter being primarily quantitative. Field studies involve researchers performing data collection in or with respect to the practitioners' actual work environment. Field studies include case studies, in which a single organization is evaluated. Given that a single organization often produces a particular type of software under unique management circumstances, results from multiple case studies are amalgamated if generalized results are desired. Surveys study practices in the field without the researcher having to be present. Formal experiments offer control of factors that often cannot be controlled in work environments: program-

ming language, program size, etc., so that the hypothesized, quantitative effects of one or more treatments can be either verified or rejected. A variation of a formal experiment that can be performed in the field is the quasi-experiment, in which subjects are not assigned randomly to treatments. A quasi-experiment may be necessary if, for example, multiple projects are being studied in the field but practitioners are already assigned to them [44]. It may also be performed when self-selected experimental design is necessary, that is, when subjects are allowed select their own treatments for ethical or advantageous reasons [36]. Case studies, surveys, formal experiments, and quasi-experiments utilize a variety of techniques for data collection, some of which, such as interviews, are applicable to multiple approaches.

An *exploratory study* is useful when the area of investigation is new or vague [36]. An exploratory study is undertaken when a researcher needs to develop a concept more fully, determine operational definitions, or uncover important variables. There is often a reliance upon qualitative techniques, although quantitative techniques can be applied as well. Unstructured and semi-structured interviewing techniques can be utilized in exploratory studies, as can participant observation, focus groups, and document analysis, all described below.

6.2.1 Data Collection

Data collection techniques are listed below as identified and categorized by Singer [145]. They are grouped into direct, indirect, and independent techniques. Direct techniques require the researcher to have direct involvement with the study participants. Indirect techniques require the researcher only to have direct involvement with the participants' work environment and artifacts. Independent techniques require only involvement with the artifacts. Within the set of direct techniques, are inquisitive and observational techniques. Of the two, inquisitive techniques, which elicit directed and undirected responses from participants, bet-

ter capture ideas and attitudes. Observational techniques are less subjective and permit accurate measurements of task durations.

1. *Direct techniques - inquisitive*
    (a) Brainstorming
    (b) Focus groups
    (c) Questionnaires
    (d) Interviews
    (e) Conceptual modeling
2. *Direct techniques - observational*
    (a) Work diaries
    (b) Think-aloud sessions
    (c) Shadowing and observation
    (d) Participant observation
3. *Indirect techniques*
    (a) Instrumenting systems
    (b) Fly on the wall
4. *Independent techniques*
    (a) Analysis of work databases
    (b) Analysis of tool use logs
    (c) Documentation analysis
    (d) Static and dynamic analysis

Brainstorming

Brainstorming is a loosely-directed elicitation of ideas from a small group of participants. It was first codified in 1953 by Osborn [119]. Its rules are easy to understand, and it is widely used in marketing, design, and task forces, less so by researchers. A moderator ensures that the participants articulate ideas that occur to them. The ideas are quickly and succinctly recorded, evaluation being deferred until a later activity. Usually performed verbally in person, electronic brainstorming is also possible. Brainstorming is effective when performed by up to twelve participants. As the number of participants increases, the number of new ideas increases, but the number of new ideas per person decreases [51].

Brainstorming is useful when seeking ideas for further exploration. It is cost-effective, as it elicits information from multiple people at once. The participants feel a sense of involvement, and they are especially motivated when they believe that the topic is important. There is a danger that a brainstorming session can become unfocused when the moderator is not well trained. There is an an added danger that some participants will be reticent to express ideas in a group setting [145]. Brainstorming is not geared toward verification or consensus.

Focus Groups

Focus groups are similar to brainstorming in that multiple participants engage in a structured discussion session, and new ideas may emerge. Patton classifies the focus group as a form of interview in which direct interaction among participants may play a large role [120]. The typical focus group includes six to ten participants having similar backgrounds. As in brainstorming, ideas are elicited for a given topic, but those ideas can be expanded, amplified, and challenged through conversation. While brainstorming often elicits radical ideas, focus groups are geared toward uncovering a consensus among participants such that outlying opinions tend to be negated. Brainstorming and focus groups are both

cost effective in that they elicit data from multiple participants. Focus groups have disadvantages over other interviewing techniques: confidentiality cannot be guaranteed, subtleties are not usually explored, the number of questions that can be addressed is constrained, and there may be reticence to express minority perspectives. In addition, it is useful only to elicit concepts that can be understood in a limited amount of time, and it requires a relatively homogeneous group of participants knowledgeable about the problem domain [86]. For example, Software Process Assessments, developed by the Software Engineering Institute, required focus groups of like practitioners: e.g., developers versus testers. [148]. Storey et al. organized a focus group consisting of programmers to validate the design of TagSEA, a navigational tool for use in software development [156]. Variations of traditional focus groups include computer mediated focus groups and electronic focus groups [86]. The former adds personal computer technology to in-person focus groups, providing real-time voting, simultaneous and anonymous contributions, group memory, and electronic record keeping. The latter provide the ability to hold focus groups across distributed geographic locations.

Questionnaires

Questionnaires are "sets of questions administered in a written format [145]." Questionnaires can be mailed or electronically distributed to a large number of geographically-separated participants, and no in-person interaction between researchers and participants is necessary, making questionnaires time and cost effective. There are no means for participants to indicate that particular questions are poorly worded or ambiguous, and researcher follow-up is required if they desire to delve into particular responses. Response rate in general for software engineering surveys has found to be about five percent [145]. Kitchenham and Pfleeger provide comprehensive guidance on performing surveys using questionnaires [82]. A survey by Lethbridge is an example using questionnaires [91]. To better understand

the areas in which software professionals felt lacking in education, Lethbridge distributed questionnaires to professionals at private companies and educational institutions. The questionnaire asked, over an array of topics, questions such as "How much did you learn about this in your formal education?," *this* being a topic whose usefulness in the professionals' careers was determined through other questions. Choices for the answers appeared within Likert scales.

Interviews

According to Patton, "we interview people to find out from them those things we cannot directly observe [120]," namely

- the interviewee's thought process, mental model, perceptions, or opinions
- historical data, sometimes to verify or explain written records, but also to capture data when the written record is sparse [138]
- clarification of observed events or actions [138]
- to garner survey information, as an alternative to a questionnaire [44].

The interviewer must know how to handle incomplete or unduly pithy responses, steer the interviewee to remain on track, and detect the need for follow-up questions. Preparation for an interview involves generation of an interview specification or guide, which can be used for interviewer training as well as direct reference during the session [82][120]. During the session itself, one or more interviewers elicit responses from one or more interviewees. Often one interviewer leads, setting direction and asking the majority of questions, while one or more others serve as scribes [138]. The interview may be audio-recorded.

Interview questions can be structured, open-ended, or interviewee-formulated [138, 145]. Structured questions, such as "how many years have you been programming?" yield quantitatively codifiable answers. Open-ended questions, whose focus is qualitative, are designed to prompt the interviewee to offer an explanation

or engage in discourse. A question such as "What is a 'requirement' to you?" [75] may result in a lengthy response, and it may uncover data not anticipated by the interviewer. Discussion points may be left the the interviewee(s), in which case the 'question,' or more precisely the discussion content, is interviewee formulated.

Interviews themselves may be structured, semi-structured, or unstructured. A structured interview consists of a fixed list of carefully worded questions asked in a defined sequence, ensuring consistency in data collected from a large number of interviewees [145]. Structured interviews often have a quantitative focus, the interviewer having specific objectives for data to be elicited. [138]. Telephone interviews are often structured: the interviewer asks a scripted set of questions, ticking or circling the evoked response or entering it as numerical data. Cost, coverage, and sampling factors help to determine whether a structured interview is advantageous over an equivalent structured questionnaire.

In an unstructured interview, "the object is to elicit as much information as possible on a broadly defined topic [137]." Patton calls this type of interview an "informal conversational interview," stating that it "offers maximum flexibility to pursue information in what ever direction appears to be appropriate [120]." The interviewer may announce a topic and allow the interviewee(s) to control the flow of conversation, or the interviewer and interviewee may be equals with respect to the flow of control. The interviewer must be trained in observing the social aspects of interviewees' behavior [44]. Unstructured segments of interviews were employed in the Software Process Assessment method for evaluating organizational software process maturity, uncovering factors that contributed to process difficulties [148].

Semi-structured interviews comprise a middle ground in which the interviewer poses specific and open-ended questions, sometimes drilling further into areas of interest based on the responses. Semi-structured interviews have an advantage over questionnaires containing open-ended questions, in that the interviewer may pose follow-up questions to delve into an area of interest exposed during the inter-

view. The quality of the collected data in a semi-structured interview is related to how well the interview is conduced [66]. Seaman provides guidance for conducting a semi-structured interview [137]. Basili and Seaman employed semi-structured interviews in a study of communication in code inspections [139]. An interview guide was constructed after each of many inspection meetings to capture information missing from the inspection's data form, and to pose follow-up questions that varied by the interviewee's role in the inspection. Audio recordings of the interviews were not directly transcribed, but they contributed to the researchers' field notes. Zannier, Chiasson, and Maurer employed semi-structured interviews to better understand how software design decisions are made [184]. Their conclusions, or 'emergent themes,' were that design is primarily about structuring the problem and that there are two disparate decision making approaches.

Experience surveys

Experience surveys are a type of semi-structured interview that can be "targeted toward discovering the parameters of feasible change [36]." The proposed change, such as the introduction of a new management practice, is evaluated in light of the subjects' prior experience and practices. It is also useful in predicting which practitioners will fit well with the change and who won't. Experience surveys may lead to the introduction or refinement of research questions or experimental parameters.

Conceptual Modeling

Conceptual modeling is a technique that can be employed within interviews or used standalone. Participants expose their conception of their mental model by drawing diagrams, designing programs, or performing other production-centered activities. Ideally, the researcher has domain knowledge of the system or process of interest, provides a framework for the drawing or other activity, and observes the

activity as well as interpreting the results [145]. Sayaad and Shirabad performed a study with conceptual modeling aimed at building a knowledge base using the terminology and concepts employed by software practitioners to describe a software system [136]. The participants organized the terms and concepts into groups and iteratively refined the emerging conceptual model.

Work Diaries

Work diaries are vehicles for practitioners to record their activities during the work day. The practitioner either records activities throughout the day, summarizing activities at the end of the work day, or noting the current activity at selected times during the day [145]. Many software consulting firms and product organizations require their personnel to complete daily time sheets which constitute a form of work diaries. An advantage of work diaries over questionnaires and interviews is that they provide data as it occurs rather than in retrospect. However, work diaries rely on self-reporting, which may be biased, and they consume time and effort. There is also a sort of 'Heisenberg Principle' of work diaries in that the act of recording may influence activity durations and sequencing. Wu, Graham, and Smith employed work diaries along with interviews and direct observation in a study of communication among practitioners in five development teams [182]. The study determined that the practitioners communicated frequently using a variety of communication modalities and changing their physical locations throughout the day.

Think-Aloud Sessions

A think-aloud protocol is a form of interview in which participants verbalize their thoughts in the course of performing an activity [120]. The interviewer's role is to elicit thoughts and feelings which are normally only internal dialogues. The interview is transcribed, audio recorded, or both. An advantage of think-aloud

protocols is the real-time rather than retrospective nature of the information, which makes it less susceptible to coloring and omission. A major disadvantage is that the participant's attention to verbal expression may alter the way in which the activity is being performed and the activity's duration, especially when prompted by the interviewer. Von Mayrhauser and Vans employed think-aloud protocols to validate their integrated software comprehension model against a variety of industrial software maintenance tasks [174]. Audio recordings of the sessions were transcribed and coded to uncover information needs during maintenance tasks, resulting in improved tool capabilities. Von Mayrhauser and Vans subsequently developed a coding scheme for analysis of think-aloud protocols wherein the highest level of granularity consists of the mental models (program model, situation model, and domain model) within their program comprehension meta-model [173].

Letovsky employed think-aloud protocols in his study of cognitive processes in software engineering [92]. Six professional program maintainers, four senior and two junior, were asked to understand a program in order to plan a modification to it. The subjects were encouraged to talk freely and also to explain why they were examining a particular piece of the code or its documentation. The researcher would prompt the subjects for explanations of their thinking process and to re-establish the verbal flow when the subject became silent. Each session was video recorded (see *Shadowing and Observation*, below). The think-aloud data were a key element in Letovsky's formulation of a program comprehension model.

Shadowing and Observation

Researchers may act as observers of individuals or groups with their knowledge, often in their day-to-day work settings. Seaman refers to this as *direct observation* [138]; Singer calls it *shadowing and observation* [145]. Direct observation allows researchers to capture quantitative and qualitative information about activities, interactions, communication, and the work environment. Researchers take field

notes which include not only what people do and say, but may also explicitly state what they do not do, for example, the omission of unit testing in a software development scenario. Field notes may also include the researcher's feelings, reactions, and reflections about what has been observed [120]. Audio recording may supplement field notes. The researcher may be detached from the group being observed or may be a participant in the group's activities, the latter scenario described in the following section, *Participant Observation*. Shadowing entails following and recording the actions of one participant at a time, while observation in general may involve many participants [145]. Limited interaction between the observer and participants may occur, especially during shadowing, to clarify an activity that's being worked on or to explain why something is being done. If such interaction is kept to a minimum, shadowing and observation can be highly cost-effective vehicles for information collection. Observation reveals high-level actions more readily than low-level details, e.g., knowing that a participant is involved in debugging, or the intermediate results of multiple-step activities, versus knowing the sequence of control keys being used.

Advantages of direct observation include cost effectiveness and the opportunity to examine practitioners in realistic settings. However, an industrial setting affords limited or no ability to control environmental variables, which may vary by location and over time. Disadvantages include novelty effects, namely the Hawthorne effect and the learning curve effect [133]. Observed performance may exceed that when unobserved due to the Hawthorne effect, in which observed parties perform better because they believe management or researchers are paying attention to them. Participants working with new techniques or tools are susceptible to the learning curve effect, the observation that people gain familiarity and facility with the technique or tool over time. Positive effects from using the new technique or tool may thus be masked early in its usage. The learning curve effect can be mitigated through adequate training and practice with the technique or tool [44]. Other

disadvantages of observation include the possibility that an observer misses an occurrence of importance, possibly due to momentary inattention, rapid pacing of events, or observer bias. This can be mitigated by employing multiple observers, each focusing on different aspects of the situation.

Singer and Lethbridge employed shadowing in a study of software engineers' work practices [144]. After administration of an initial web-based questionnaire, the researchers shadowed a software engineer new to the development group one to one and a half hours per day. After six months, shadowing was reduced dramatically, as redundancy was prevalent in the sessions. The researchers were able to characterize the engineer's events, interactions, and percentage of time spend in different activities, detailed to the level of knowing that issuing a Unix command was the most frequently-repeated action. To capture detail, the researchers utilized *synchronized shadowing*, wherein two observers simultaneously shadow an individual engineer, merging the two sets of field notes afterward.

Letovsky employed observation in a controlled setting during his study of cognitive processes in software engineering [92]. Subjects were videotaped in conjunction with think-aloud protocols while working on a program comprehension problem, providing the researcher visual data to supplement and corroborate the verbal data.

Participant Observation

The researcher may act as a participant in the group activity being observed, a situation which Singer, Seaman, and others refer to as *participant observation* [138][145]. The researcher is able to capture interactions from a first-person point of view instead of as an independent observer. The researcher may be unable to take field notes during the observed activities, and it may be difficult to retain objectivity. The researcher must gain the trust of the group and mitigate the risk that group members are not continually self-conscious of being observed. Because

of the time needed to gain trust and comfort with the group, the duration of participant observation is typically a few weeks or more [44].

McAvoy and Butler performed a qualitative study of learning in an agile software development team [104]. Their primary approach involved eight months of participant observation within a team of seven developers. Their focus was the failure of translating "espoused theories" or values into actual practice. Seaman and Basili utilized participant observation in a study of communication in software inspections [139]. Seaman was embedded in a software development team that performed formal inspections over a period of a year and a half. Participant observation was supplemented by interviews. The overall study invoked both quantitative and qualitative methods for analysis of the data. Seaman's observations inductively produced well-supported hypotheses concerning the relationship between the network of relationships between developers and the quality of inspections, including the relative time spent in different inspection sub-activities. This study is an exemplar of data collection techniques from the interpretivist tradition being applied under a conventional mind set.

Indirect and Independent Techniques

A number of data collection techniques involving indirect interaction with subjects or practitioners can be used as alternatives or supplements to those described above. These techniques are summarized by Singer [145]. System instrumentation such as keystroke and event recording can capture human-computer interactions for many participants over a long period of time. *Fly on the Wall* is a hybrid technique in which participants video or audio tape themselves, allowing the researcher to be absent during data collection. Work databases such as configuration control repositories can serve as data sources for analysis, as can logs of automated program building tools such as Ant [165]. Documentation as a data source includes source code comments as well as specifications and other technical documentation

independent of the code. Successive snapshots of work databases and documentation over time can offer insight into the evolution of a software product. Finally, static and dynamic analysis of the software product itself can offer insight into the developers' thinking, especially about the product's architecture and organization.

6.2.2 Coding, Analysis, and Experiment Design

Quantitative coding and analysis involve well-established techniques whose explanations are readily available in the literature along with guidance on designing experiments [36][103]. Experiment design and analysis with a slant toward software engineering is presented by Basili [7] and by Pfleeger [123].

Validity of statistical significance tests in an experiment is maximized through randomization, the random assignment of subjects to groups and of treatments to experimental objects, which ensures independence [123]. If the treatment is an enhancement to a program comprehension tool, then a randomized block design is appropriate, in which each developer is assigned to one of two groups, one using the enhancement and the other not. Moreover, multiple programs should be used to prevent localized effects based on a single experimental object, resulting in a four-group design. The number of subjects may be reduced by having each subject use the baseline tool on one program and the enhanced tool on the other program, as shown in Table 6.1. The experimental objects and treatments are ordered, allowing analysis to reveal a learning curve effect if present.

Group	First Program and Treatment	Second Program and Treatment
1	Program 1 with baseline tool	Program 2 with enhanced tool
2	Program 1 with enhanced tool	Program 2 with baseline tool
3	Program 2 with baseline tool	Program 1 with enhanced tool
4	Program 2 with enhanced tool	Program 1 with baseline tool

Table 6.1: Treatment groups

Qualitative coding and analysis are presented as an entire section in the book by Patton [120]. Coding in qualitative studies consists of marking up pieces of

text collected through interviews, surveys, and other approaches to uncover concepts. Inductive analysis is performed to discover patterns, themes, and categories. Data may also be analyzed through deductive analysis, in which they are analyzed against a known framework. Qualitative analysis is typically inductive in earlier research states and deductive at the latest stage. An inductive approach to coding called *open coding* has been codified by Strauss and Corbin, whose comprehensive coding and analysis methodology is one of the major forms of *Grounded Theory* [158]. In grounded theory, higher-level concepts are successively discovered from analysis of lower-level concepts, forming a hierarchy. The highest level represents a small number of themes, possibly a single theme, about the culture or group being studied. Preformed codes, which the research formulates prior to coding, are verified or rejected, and they may result in lower as well as higher level codes. Divergence as well as convergence in coding and classification are notable. Divergence may uncover outliers which might serve to extend a formative theory or challenge the explanations for convergence. While coding and analysis proceed, the researcher may also formulate hypotheses that can be tested through further study. Ideally, coding occurs while data collection is still in progress, so that the researcher can determine when saturation has occurred and data collection can thereby stop. Seaman further discusses the application of grounded theory in software engineering research [137].

Adolph, Hall, and Kruchten have described how grounded theory can be employed to study the practice of software development [1]. Use of grounded theory is appropriate for such study, they maintain, if we accept the idea that people trump process, so that observation and interviewing rather than the process documentation yields a true idea of actual practices. The authors point out that most articles purporting the use of grounded theory only minimally employ the methodology. They summarize their experience performing yet-to-be-published studies in the form of fifteen guidelines, the fifth of which clarifies the somewhat vague ideas

of concept, indicator, category, and property. In the eighth guideline, they recommend participant observation as a first class data collection strategy on par with interviewing. Detailed note-taking and phased research are also emphasized. Guideline 14 states that commonly used rigor criteria for quantitative research are not useful for judging the quality of qualitative research. Representativeness of theory to the data and trustworthiness of the data are suggested criteria. While some of the guidelines are specific to rigorously-employed grounded theory, others such as Guidelines 8 and 14 appear to apply to qualitative research in general.

Grounded theory was employed in a study of software process improvement in Irish software product companies [35]. Findings included the discoveries that all of the companies tailor their standard software processes, that process formation depends on the background of the software development manager, and that verbal communication was substituted for documentation to reduce documentation costs, resulting in increased sharing of tacit knowledge.

Data from multiple sources, whether quantitative or qualitative, may be combined during analysis, a practice known as triangulation [138]. Triangulation is employed in several of the software engineering studies cited above, including the inspection study by Seaman and Basili [139]. One form of triangulation, known as replication, occurred in the inspection study, as patterns of data occurred in multiple data sources over many software inspections. Berling and Thelin employed triangulation to analyze data from interviews, documentation, and direct observation in a case study of the verification and validation process in a software development organization [9]. The study centered around the tradeoffs between inspections, which occur early in the development process, and testing, which occurs late. Accordingly, Berling and Thelin developed a goodness measure to compare the time into a project at which a fault is found with the time it hypothetically can have been found.

6.2.3 Evaluation Frameworks versus Empirical Techniques

Software sonification is an audio analogue of software visualization, in that aspects of software systems are represented aurally or visually, respectively. Software visualizations have been evaluated by constructing user scenarios and walking through them to see how well they meet the criteria put forth by various question-based evaluation frameworks, such as that elaborated by Storey [157] and Knight [83]. Kitchenham points out the preliminary nature of such a process, calling it *qualitative screening* and indicating that it can be initially performed by examining the tool's literature without actually using the tool itself [81]. Evaluation frameworks in the sonification arena are neither mature nor widely accepted, the focus being more toward an experience-based, design pattern approach [49]. According to Smith, empirical studies of visualizations "attempt to quantify the benefits of visualizations and provide hard evidence about some hypotheses [146]." Empirical evaluations of software visualizations have been undertaken by Wiss and Carr [181] and Storey [155]. The empirical approach requires more resources, but it is more comprehensive, and "it can highlight usability issues that may have been overlooked by other methods [146]." A disadvantage of empirical studies of sonification is that the sonification, and software sonification in general, is likely to be entirely new to practitioners, so that their strategies for using a sonification tool and their performance will differ from that of someone who has been using the technique for some time and has a high comfort level with it. One way to overcome this disadvantage would be to study a group of participants using the technique and tool regularly over a long period of time (weeks or months). A more practical way to partially overcome this disadvantage is training that includes hands-on experience, reinforcement, and feedback.

6.2.4 Chosen Approaches

Because both the idea of sonification in comprehension of static software structure and the particular sonification design employed are new, it is appropriate to conduct an exploratory study as a check of the sound mapping scheme's viability and as a vehicle for suggested improvements, followed by a formal experiment to draw conclusions about its effectiveness in use. The exploratory study is preceded by even earlier feedback in the form of brainstorming and other informal interaction to help formulate the sound mapping. The exploratory study itself uses mainly qualitative techniques, while the formal experiment quantitatively addresses a selected hypothesis concerning the sound mapping and tool. Interviews and direct observation of subjects, though expensive in terms of researcher time, are vehicles which promise to provide much information. Pre-trial questionnaires provide subject demographics and experience profiles. A post-trial questionnaire, at minimal cost to the researcher and subjects, may capture supplemental information about the experience of using the tool.

6.3 Summary

This chapter has examined empirical approaches in software engineering. The following chapter sets forth the approaches and results of both the exploratory study and the subsequent formal quantitative study.

Chapter 7

Studies and Results

7.1 Introduction

Two human-subject studies were performed using the reference sound mapping, the first exploratory and primarily qualitative, and the second a quantitative experiment. This chapter describes the design and results of both studies. Beyond this Introduction, the chapter is divided into two major parts, Section 7.2, which describes the methods for both studies in turn, and Section 7.3, which presents the results of both studies. The chapter concludes with the Summary, Section 7.4.

7.2 Methods

7.2.1 Study One Method

The exploratory study has the following goals:

1. Validate that the mapping concept is viable - that the reference mapping can be easily learned, understood, and retained.

2. Uncover areas for improvement in the mapping.

3. Discern if the mapping's projected usage is viable.

4. Obtain further ideas for the mapping.

5. Obtain further ideas for using the mapping.

6. Formulate guidelines for this and alternate mappings.

Strategy

Subjects participated in one-on-one sessions in which they listened to the reference sound mapping and offered semi-directed feedback, then discussed the tool and how they might use it. The study had three phases: participant selection, participant sessions, and analysis, described below. The protocol for participant sessions appears in Appendix A.

Participant Selection. The study was designed to achieve saturation, which occurs when successive subjects reveal little or no new useful information [158]. Starting with an initial number of subjects, additional subjects are added until saturation occurs. In the present study, the number of subjects is deemed sufficient when the sound mapping's viability or lack thereof is established and the flow of new ideas become minimal. The initial number of eight subjects was based on the homogeneity of the population (all software professionals), the well-bounded problem domain, and the informality of the study.

Potential subjects, candidate users of the tool in practice, were required to have knowledge of object-oriented software design principles. Subjects were professional software developers and computer science students. Full-time academics were permitted as long as they had served in industry in the past. Students may be no less than 4^{th} year upperclassmen due to the requisite programming experience. No musical skill was required.

Participant Sessions. Each individual-participant session was structured to last no more than two hours. Participants were permitted to withdraw from the session at any time. Participant were briefed verbally and in writing on the session to take place, and each signed a consent form. To begin each session, the

participant was questioned to verify their object-oriented programming knowledge, and minimal testing was performed to determine the participant's level of musical listening acuity. This was followed by in-session training on the audio mapping. The core of the session consisted of observations of and directed feedback from the participant while listening to audio streams representing mappings from actual software projects, followed by discussion of the mappings and the tool. Those four activities in sequence, determine skills, train participant, listening and obtaining participant feedback, are described below.

1. Determine Skills.

Each subject's basic understanding of object-oriented software construction was verified. The subject was asked if he or she had worked with a third-generation, object-oriented language, notably Java or C#. The subject was queried as to their understanding of interfaces, static methods, and accessor/mutator methods. The subject was required to demonstrate a basic level of understanding to continue the session.

Each subject was also given a minimal audio understanding test using a pre-recorded audio stream. First, pairs of tones were presented, with the subject answering which tone is higher in pitch, or if both tones are the same pitch. The test progressed to multi-tone sequences. Finally, tone sequences were presented at different simulated distances and azimuth angles within a stereo image, the subject being asked to determine whether each tone was near or far and left, right, or centered. *participant_test.wav* is the stream that contains the test items.

2. Train Participant.

Each subject was asked to listen to a fourteen-minute, pre-recorded training stream that introduced the sound mappings and sequential sound presentation. The audio file *training_stream.wav* contains the training stream. Subjects were given the opportunity to pause the stream and replay it in part or in its entirety. Subjects were also afforded the opportunity to pause the stream at any time to

obtain clarification. The stream is based in part on a software program called *Simple*, containing two packages, three classes, and an interface shared by two of the classes. The subject was shown the *Simple* project as it appears in an Eclipse Package Explorer. The subject was also introduced to the Sonify View of the tool and shown how sequences of entity-sound mappings would be played. The subject was given, as a reference, a tabular summary of the general entity-mapping scheme divorced from any particular software program.

3. Listening.

Five sound streams drawn from the *Expenses* software program were played for each subject, the subject being asked to answer various questions and reproduce some of the program's static structure on paper. Subjects were encouraged to engage in a mediated think-aloud protocol in which areas of confusion and other observations were articulated. The subject was also observed to help assess the relative ease or difficulty at each point. The investigator reserved the ability momentarily pause the stream to further assess understanding or determine what specific difficulty was being encountered.

Expenses is a snapshot of an incomplete expense tracking program written for this exercise. The snapshot compiles to a working prototype that affords prompted, text-based entry of personal expenses. Information about each expense, such as its amount and whether it is taxable are entered and stored in a delimited text file. The method that stores the information inherits from an abstract method, as does a stub method intended to eventually implement alternative data storage in an XML format. The text-based client is separated from processing and storage functionality so that, in the future, a graphical client can be added. The program is implemented within a single Eclipse project, the client, server facade, and server-side data storage functionality belonging to separate packages. External libraries are referenced, notably *java.io* for both screen I/O and file storage. Visual or aural examination of the project should easily reveal the

existing inheritance and polymorphism, the idea that each package represents an architectural layer (client, facade, and input/output), and the I/O-centric nature of the input/output package and its subordinate entities. The five sound streams are described below.

expenses_5sec.wav - Serially-expressed parent-child relationships of all packages, classes, interfaces, and methods in the Expenses project. Adjacent entities are separated by a 5-second pause. The subject is asked to classify entities, identify entities, describe entities' characteristics, and describe the project's structure.

expenses_2sec.wav - The same serialization as expenses_5sec.wav, but adjacent entities are separated by a 2-second pause. Subjects were asked to classify objects and describe the projects' structure.

called_by_ExpenseFacade_constructor.wav - Serialization by parent-child relationship of the entities called by the constructor ExpenseFacade(). The subject is asked to classify entities and describe their characteristics, including whether they are part of the local project or an external library. All entities in the stream reside inside the local project.

called_by_ExpenseDelimitedAccessStore.wav - Serialization by parent-child relationship of the entities called by the ExpensedelimitedAccessStore() method. The subject is asked to classify entities and describe their characteristics, including whether they are part of the local project or an external library. Several entities in the stream reside inside the local project, the rest residing in the *java.io* package external to the project.

class_size_ascending.wav - Classes of different sizes. Individual classes from this stream are played, and the subject guesses which size range the class belongs to.

4. Obtain Participant Feedback.

After the sound stream presentations, each subject participated in a semi-directed interview to obtain their observations and impressions. Each session continued until the subject had nothing further to say. Interviews were audio-recorded.

Post-Session Analysis. After participant sessions, performance data from the listening exercise were analyzed to uncover which kinds of mappings and sequences posed greater or less difficulty, whether trends existed in understanding based on the mappings, and whether level of programming experience or musicality had an impact. Feedback was analyzed using shallow open coding to uncover common reflections, ideas, and criticisms. Notable individual thoughts were identified, as were outlying listening responses. The session-specific results were amalgamated to improve the mapping, formulate guidelines for mapping non-speech sound to software entities and inform the second study, and begin to catalog recommendations for future research.

Limitations

It is emphasized that Study One was exploratory. Neither the number of subjects nor the experimental design was sufficient to measure the efficacy of any given sound construct as deployed in a tool in an everyday situation to a high level of confidence, and the study was neither intended nor designed to achieve such a result.

Internal and external validity threats are summarized in Table 7.1 and discussed in the list below.

1. *Inconsistencies among sessions.* It is possible that the relationship between researcher and subject varied among sessions, inducing differences understanding the sound mapping due to comfort level and differences in qualitative feedback due to unobserved prompting.

Limitation	Category
Inconsistencies among sessions	internal
Methodology	internal
Acoustical environment	internal
Possible subject selection bias	external
Single program	external
Single mapping	external
Cultural limitations	external

Table 7.1: Study One internal and external validity threats

2. *Methodology.* Quantitative techniques are typically employed for concept discovery. The data are coded and analyzed in layers to uncover categories and new concepts. In Study One, textual data were analyzed for commonalities, ideas, and opinions, but not to uncover hitherto unknown categories or major concepts. Coding was flat, restricted to one layer above the textual data. Subjects' abilities to understand and recall the sound associations were observed informally rather than being measured using rigorous quantitative means.

3. *Acoustical environment.* The acoustical environment varied by session, as most of the sessions occurred at a location convenient to the participant. This may have caused aural recognition to vary among participants.

4. *Possible subject selection bias.* Subject sampling, while targeted to achieve diversity in non-speech auditory sophistication and software development experience, was likely biased most subjects having been software professionals previously known to the investigator. Subjects were largely of the same cultural background as the investigator, meaning that metaphors employed in the sonification scheme, and validated by subjects, cannot be assumed to be culture-independent. Several subjects were not native English speakers but have lived in the U.S. for at least five years.

5. *Single program.* The single program utilized may not be representative of

any significant subset of programs, notably because it is an early prototype.

6. *Single mapping.* Because only one mapping was utilized, it is unclear which mapping decisions need to be present to determine an equivalent class of understandable mappings.

7. *Cultural limitations.* The meanings of sounds may be culturally influenced such that the results cannot be generalized to all cultures represented in the software engineering community.

7.2.2 Study Two Method

The second study was a quantitative experiment involving twenty-four software professionals and advanced university students performing comprehension tasks via code-reading and listening, using Eclipse along with its sonification extension. The study's intent is to evaluate task performance with and without sound. The experiment addresses the following research question:

> Can an integrated development environment (IDE) outfitted with sound to depict characteristics of a program's static structure facilitate program comprehension when compared to the same IDE instead outfitted with equivalent capability to visually search for method references?

Facilitate in this context means to reduce time or effort or increase correctness. The null hypothesis is

> H_0. Use of a sound mapping of static program structure during comprehension tasks does not reduce the time taken to perform the task.

The alternate hypotheses, below, follows from the null hypothesis.

> H_A. Use of a sound mapping of static program structure during comprehension tasks reduces the time taken to perform the task.

The hypothesis is concerned with measuring task duration. Effort, while related to task duration, also encompasses cognitive load and the actions necessary to complete the task. Effort is treated in an explanatory manner. Task correctness is treated as an explanatory covariate.

Population and task characteristics captured in the experimental scenario are summarized in Table 7.2.

Characteristic	Description
population	Professional software engineers and advanced computer science students. Experiment is designed such that differences between professionals and students can be detected.
engineering role	quality assurance or validation & verification
software maintenance type	perfective
language	Java (1.4 and beyond)
programming environment	Eclipse (3.3.1.1 and beyond)
program size	300 to 1,000 lines of code

Table 7.2: Population and task characteristics in the experimental scenario

Experiment Design

The study is a 2 x 2 crossover experiment. Each of the 24 subjects performed two sets of tasks, each set focusing on one of two computer programs. One set of tasks was performed using sound, the other without. The 24 subjects were allocated to four groups, six subjects per group, to which the programs and treatments were administered as shown in Table 7.3. Software professionals and university students were members of all four groups in similar proportion.

The minimum number of subjects needed was determined using the Power Analysis and Sample Size (PASS) computer program [113], which was given as input the experimental parameters and a desired significance level of 0.05 or less.

Group	First Program	Second Program
1	CP (sonified)	PICT (unsonified)
2	CP (unsonified)	PICT (sonified)
3	PICT (unsonified)	CP (sonified)
4	PICT (sonified)	CP (unsonified)

Table 7.3: Treatment groups

Population to be Studied

The population consists of programmers with working knowledge of Java who have no significant visual or aural impairment. Professional and advanced student programmers were studied, as both groups possessed the capability to adequately understand software structure at the level sonified in this study, and both were able to perform the required tasks.

Subject Selection

All of the software professionals had experience developing production-quality software subsequently placed in public or private use. Some were full-time software developers in government and industry, while others were employed in academia but had prior full-time experience in industry. Persons having only academic experience were excluded. The students were upper-level undergraduates and postgraduates in programming-intensive computing disciplines, all of whom had developed production-quality programs as projects, as demonstrated by placement of their programs into service. The subjects were familiar with Java and conversant with Eclipse, JGrasp [74], or similar software development environments.

Subjects were recruited in two locations: Baltimore, Maryland, USA and Melbourne, Florida, USA. Professionals in both locations were recruited from area companies, universities, and government agencies. Students were recruited from Loyola University Maryland, the Johns Hopkins University, and the Florida Institute of Technology, specifically from the graduate and undergraduate computer science programs. Subjects participated entirely on a volunteer basis and neither

expected nor received compensation. The minimum qualifications for subjects are listed in table 7.4.

	Professionals	Students
academic level	undergraduate degree	upperclass undergraduate
major	any	computer science
programming knowledge	three years object-oriented third-generation language	three years object-oriented third-generation language
Java knowledge	see text	see text
Eclipse knowledge	desirable but not necessary	desirable but not necessary
impairments	no significant visual or hearing impairment	no significant visual or hearing impairment

Table 7.4: Minimum subject qualifications

Two graduate students participated as subjects in a full pilot version of the study. Adjustments based on observation and extensive pilot-subject feedback were incorporated into the actual study. The resulting in-person training protocol is included in Appendix B, and the experimenter's instructions for executing trials are included as Appendix C.

Experimental Objects

The two Java programs selected as experimental objects were chosen on the basis of their size and their ability to undergo improvements whose characteristics can be determined by a single person within a reasonable task duration (under fifteen minutes). Program size (or size of a well-bounded subset of the program) and complexity were constrained so that, again, task duration would fall within reason, while complexity was high enough to possess a real-world rather than "toy" set of capabilities.

The selected programs, Course Predictor (CP) and a Pictionary emulator (PICT), had been produced in an academic environment. They are functioning, production-quality programs, yet they afford clear opportunities for perfective maintenance. For example, inconsistent implementation of log files in CP can be discovered in a short period of time, compared to implementations in larger and

mature programs found in open-source repositories such as Sourceforge [151], due to the size of CP and the unambiguous nature of its logging code.

CP, or Course Predictor, is a Java program written by a senior faculty member who specializes in software engineering. Written as an applet, it has been in production for several years on a public web site. In a later version, it had been converted to an application for retargeting and inclusion of logging capabilities by subsequent programmers. The application version is a work in progress whose logging capability is in need of streamlining.

PICT is a multi-player, word-guessing game, modeled after the board game Pictionary[2] and implemented in Java as a group project during an upper-level undergraduate Software Engineering course. The program was specified, implemented, and formally tested in conjunction with a business sponsor. After completion, the game was played heavily in group demonstrations without encountering errors. In the game, each player interacts with their own instance of a visually-oriented client. A server manages player turns and keeps score. It also provides words in turn, in randomized order, from a pre-populated word list. At the beginning of the game, the word list file is read in and randomized by the server. At the beginning of each turn, a player is presented a word from the list and then produces a drawing in a pixel field using the mouse. The other players see the picture as it is drawn and attempt to guess the word, typing their guesses into a text box. The turn ends either when the word is guessed or the allotted time expires. The server and clients communicate through TCP/IP sockets. Only the server-side code was used in the trial.

Characteristics of PICT and CP are given in Table 7.5. *Lines* indicates the number of text lines in the program, while *LOC*, or lines of code, indicates the number of lines of source code as determined by an Eclipse Metrics plugin [135].

A third program, EXPENSES, previously described in Section 7.2.1, was used for training purposes immediately prior to each subject's trial.

Program	Packages	Classes	Interfaces	Methods	Lines	LOC
PICT Server	1	3	0	10	489	351
CP	1	7	1	31	953	665

Table 7.5: Program characteristics

Feature Subset

The reference sound mapping was manually realized for each program, as automated realization of sound mappings from a program has not yet been implemented. A subset of the sonified relationship types in the reference mapping was made available to subjects, due to the time and effort involved in manual sound mapping realization.

Having selected an entity within the Package Explorer, each subject was able to select and listen to any of the following realizations:

- the selected entity itself
- the class that the selected class extends
- interfaces that the selected class implements
- entities that instantiate the selected class
- entities that call the selected method

The omitted relationship type was

- entities *referenced by* the selected entity

The omission reduces the range of sound-related actions available for performing the tasks, but it also decreases the time needed for training.

Eclipse releases since at least 3.5.1 have included a Java search feature whereby a developer can select a Java entity and invokes a search for its references. To do so, the developer completes a search form, and the results appear in a search results tab in the lower right portion of the Eclipse window. As seen in Figure

7.1, for the CP program, the search results tab shows that *CLog* is instantiated by the class *cpred* as the variable *consoleLog*. Subjects not already familiar with Java search were introduced to the feature during the in-person training period. Subjects were allowed to use the Java search feature only while performing tasks in the unsonified environment. Otherwise, subjects were permitted to use any other feature provided by Eclipse for task performance under either treatment.

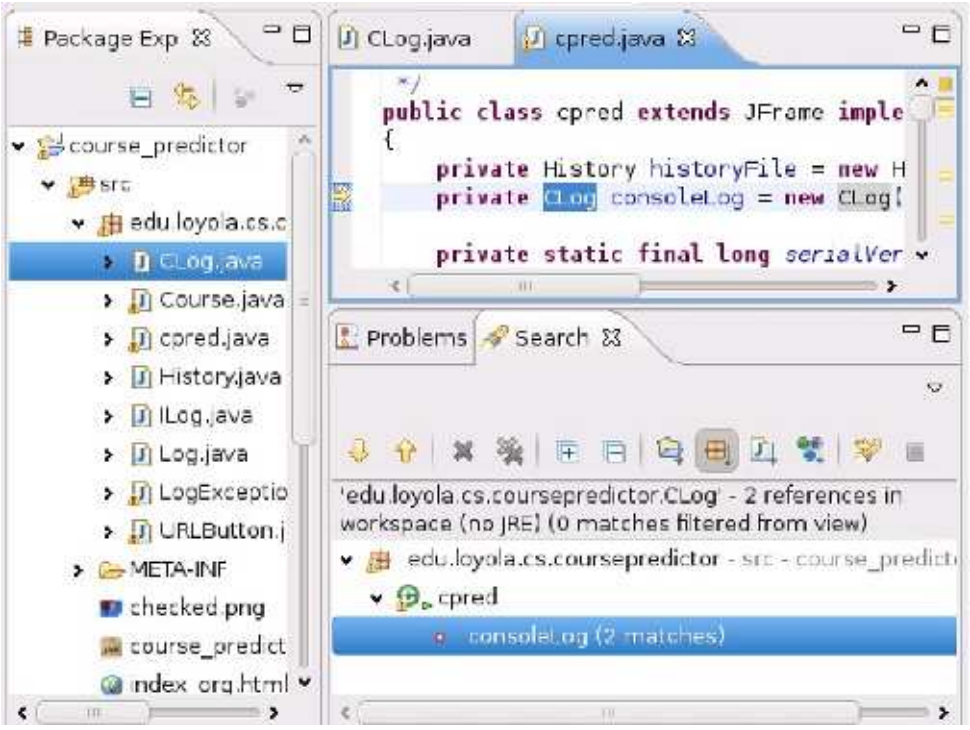

Figure 7.1: Java search feature of Eclipse

Protocol

The experiment's flow, from the subjects' perspective, is depicted in Figure 7.2. Subjects were instructed to perform training activities and a questionnaire two to three days prior to the trial. Each subject was asked to download and listen to a twelve-minute audio training stream, followed by a brief set of listening exercises. Each subject also completed an ethics form and a questionnaire concerning his or her software experience, musical experience and training, and demographic profile. The trial itself was an individual session lasting approximately 90 min-

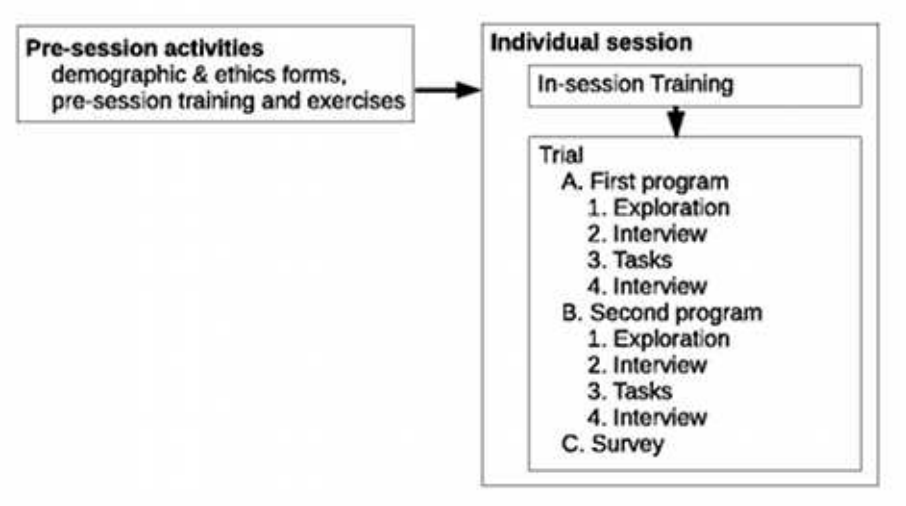

Figure 7.2: Experiment flow, from subject's perspective

utes. To begin the session, each subject received additional, hands-on training, performing tasks on the EXPENSES program using the Eclipse sonification extension. Brief interviews were conducted immediately after exploration to capture the exploration strategy. The subject then explored and performed tasks on each program in turn. Brief interviews were again conducted immediately after the last task for each program. The interviews were audio recorded.

Subjects was asked to freely explore the program for a specified time period: 12 minutes for CP and 5 minutes for PICT, using or not using sound according to the subject's group assignment. The subject then performed five tasks per program, again sonified or unsonified according to the subject's group. The first four of the five tasks per program, CP 1 through 4 and PICT 1 through 4, require only a simple lookup strategy to answer straightforward questions. CP 5 and PICT 5, both perfective maintenance tasks, require a more complex information assimilation strategy. Subjects were asked to explain why the program does not meet a new set of requirements or how it might be streamlined. All of the tasks are reproduced in Tables 7.6 and 7.7 along with their solutions and maximum allotted durations.

The final tasks for each program, CP 5 and PICT 5, were used to test the

Task	Task Statement	Solution	Max. Dur.
1	Which package/class/method combinations instantiate the class URLButton?	The method cpred.init.	5 min.
2	Identify all classes and methods which are callers of the method cpred.greenPanel.	The method cpred.init.	5 min.
3	Does URLButton implement any interfaces? If so how many? Are they internal or external to the Java project?	It implements ActionListener, which is external.	5 min.
4	Is URLButton.actionPerformed called by any code internal to the project? By any code external to the project?	It is called by the infrastructure.	5 min.
5	Previous developers have implemented logging of desired messages. They have each worked on their own logging code, so we know that it can be streamlined. Currently, logging may or may not meet the following requirements: (1)All messages that are logged will be logged to both the console and a log file. (2) All logging shall occur via a single logging class within the project. (3) All logging shall utilize the built-in Java class java.util.logging.Logger. Determine if the project meets the requirements stated above. if they are met, how? If not, why? How can logging be streamlined?	CLog logs only to the console, and it doesn't use the Logger class, but multiple cpred methods call it. Log uses the Logger class, and one method calls it. History extends Log, albeit improperly, and it is instantiated, but it is never subsequently called for any logging. To streamline, remove History and its instantiation, remove CLog, and redirect the CLog calls to instead call Log.	15 min.

Table 7.6: CP Tasks

Task	Task Statement	Solution	Max. Dur.
1	Identify all callers of the method WordRepository.getNextWord.	GameServer.run.	5 min.
2	What instantiates WordRepository?	The constructor in the GameServer class.	5 min.
3	Identify all data writers within the Pictionary Server package.	There are none.	5 min.
4	Identify any/all dead classes and methods, i.e., those that are never called. Write the answer on paper.	getCurrentWord, getList, getNumberOfWords, and getWordsLeft.	5 min.
5	Ensure that the code meets the following two design criteria: The entire word list will be made available in randomized order (a) before the first round is begun, and (b) after every five rounds of turns. (This is an easy way to minimize word repetition while avoiding running out of words.) For each of the two criteria, if the code does not meet it, explain why not. If the code meets it, explain how.	The word list is only refreshed upon initialization, so only (a) is met. WordRepository.shuffleWords is called by WordRepository.loadFile, which in turn is called by the WordRepository constructor, which is only called by the GameServer constructor, which is only called by GameServerMain.main.	15 min.

Table 7.7: PICT Tasks

experimental hypothesis. The first four tasks for each program were included to afford the subjects familiarity and practice with the Eclipse environment, the sound mapping, and the visual search function. All ten tasks were observed and timed.

Study Limitations

Internal and external validity threats are summarized in Table 7.8 and discussed in the list below.

1. *Acoustical environment.* The acoustical environment varied by session, as most of the sessions occurred at a location convenient to the participant. This may have caused aural recognition to vary among participants.

Limitation	Category
Acoustical environment	internal
Restriction of available features	internal
Task sequencing	internal
Familiarization with the sound mapping	internal
User interface limitations	internal
Measurement criteria	internal
Program selection	internal
Task selection	internal
Possible subject selection bias	internal, external
Applicability to other programming languages	external
Applicability to the visually impaired	external
Applicability across cultural boundaries	external
Accuracy	internal

Table 7.8: Study Two internal and external validity threats

2. *Restriction of available features.* As pointed out above, the ability to select an entity and hear those entities that it calls was omitted. This restricts a means to investigate relationships to off-screen entities, which may in turn impact task completion times. As mitigation, tasks were selected which do not rely upon the omitted feature.

3. *Task sequencing.* The sequencing of four simple tasks followed by a fifth, more complex task may lead to learning effects, which may impact task completion time and correctness. This is in contrast to possible learning effects between the two parts of the trial, which is mitigated through the ordering of the parts by group.

4. *Familiarization with the sound mapping.* Most subjects were introduced to the sound mapping, and indeed to the entire concept being tested, only by performing the take-home training one to two days prior to the trial, supplemented by the in-person training immediately prior to the trial. Two subjects had been exposed to the sound mapping in the previous study, however, over a year separated the two studies. In actual use, subjects'

abilities to retain the meanings of sound mappings and develop strategies using sound would be expected to noticeably improve after having had weeks or possibly even a few days of experience using the tool, especially after having worked repeatedly with the same Java program.

5. *User Interface Limitations.* The Eclipse sonification extension's user interface has several limitations that may lengthen task durations. First, a sequence of entities cannot be interrupted; it must play until complete. Second, user interaction is blocked during sound playback. Third, the scheme whereby one selects an entity in the browser, selects the desired related entities to play, and clicks Play may not be optimal compared to playing the sound on mouse hover, which is envisioned for future versions of the user interface. The visual Java Search feature, on the other hand, has been optimized over many production releases of Eclipse.

6. *Program selection.* The programs selected to serve as experimental objects are relatively small in size for production programs, though not trivial. Moreover, they may be more representative, in terms of structure and complexity, of programs in the academic domain as opposed to other domains (e.g. medicine, defense, business).

7. *Task selection.* The subject is expected to analyze the source code but not perform any actual programming. This presents the possibility that applicability better fits tasks of a quality-assurance, verification and validation nature than hands-on programming tasks.

8. *Possible subject selection bias.* It is possible that the sample is biased in terms of skill set and level, due to the small number of organizations from which subjects were recruited. There is also the possibility of sample bias toward musically-oriented members of the population. That is, it is possible that musically-oriented persons were be more interested in par-

ticipation than those less musically oriented, due to the nature of the experiment, and are therefore over-represented. Subjects were largely of the same cultural background as the investigator, and metaphors employed in the sonification scheme cannot be assumed to be culture-independent. All subjects spoke English fluently, but several were not native English speakers. Musical-orientation bias is mitigated through recruitment of subjects by non-musically-oriented agents, blind recruitment via physical and electronic announcements, and recruitment of subjects in group situations.

9. *Applicability to other computer languages.* The experiment is restricted to the Java programming language. Results and conclusions should, in general, be applicable to other third-generation, object- oriented programming languages, as they are similar to Java. Some languages in this class are C# [108], C++ [160], Ada95 [163], and Python [98]. Applicability to languages outside that class is unknown. Alterations to the sound mapping scheme would be necessary for those languages.

10. *Applicability to the visually impaired.* Visual navigation and reinforcement restrict results of this study to the sighted, although it is expected that the aural cues would be processed at least as well by the visually impaired.

11. *Applicability across cultural boundaries.* The sound mapping consists of arbitrary as well as suggestive sounds. The interpretation of meaning in the sounds may differ across cultural boundaries.

12. *Accuracy and Precision.* Timings are rounded to the nearest second. Accuracy may be affected by the manual nature of the timing.

Measurement, Data Collection, and Data Storage

Timings were performed manually using a stopwatch. The subject signaled completion of each task prior to offering an answer or explanation. The experimenter

paused and restarted the stopwatch when posed with a procedural question or request for clarification.

Task correctness was measured as incorrect, partially correct, and fully correct. For a task to be partially correct, the subject must demonstrate a level of understanding that substantially leads toward the full solution. In particular, CP Task 5 in Table 7.6 lends itself to partial correctness.

Subjects were interviewed as to their exploration strategies immediately after the free exploration period for each program. Subjects were also interviewed after performing tasks CP 5 and PICT 5. Responses were voice recorded.

The IBM Post-Study System Usability Questionnaire (PSSUQ) [93] was presented to subject immediately after the trial to provide feedback about the usability of the Eclipse sonification extension. The instrument contains 19 questions requiring responses on a seven-point Likert scale, 1 meaning strongly agree and 7 meaning strongly disagree, with space for free-form comment. As the questionnaire is meant to be applied to an entire system or application, it was emphasized to the subjects that, in this case, it applies only to the sonification extension. Question 9, having to do with error messages that are minimal in the current tool implementation, and questions 11 through 14, having to do with non-existent help and documentation, were categorized as not applicable (NA). In addition, subjects were allowed to answer NA to any question for which they had no opinion or could not answer. The questions are listed below.

1. Overall, I am satisfied with how easy it is to use this system.

2. It was simple to use this system.

3. I could effectively complete the tasks and scenarios using this system.

4. I was able to complete the tasks and scenarios quickly using this system.

5. I was able to efficiently complete the tasks and scenarios using this system.

6. I felt comfortable using the system.
7. It was easy to learn to use this system.
8. I believe I could become productive quickly using this system.
9. (NA)
10. Whenever I made a mistake using the system, I could recover easily and quickly.
11. (NA)
12. (NA)
13. (NA)
14. (NA)
15. The organization of information on the system screens was clear.
16. The interface of the system was pleasant.
17. I liked using the interface of the system.
18. This system has all the functions and capabilities I expect it to have.
19. Overall, I am satisfied with this system.

Task timings and observations were placed in a folder per subject along with the completed pre-trial questionnaire, trial-time researcher notes, any trial-time artifacts produced by the subject (such as handwritten answers to the question posed in PICT Task 4), and the PSSUQ responses. Data were later transcribed to a spreadsheet and exported to *R* [166], a statistical analysis program, for analysis using tests of statistical significance.

7.3 Results from Studies

This section reports the results from both studies and also captures early feedback that helped to shape the sound mapping.

7.3.1 Study One Results

This subsection describes the results of the exploratory human-subjects study, performed during the period June through December 2009, from subject selection through listening results.

Subject Profile

Ten subjects were required to reach saturation. The subjects exhibited diversity in sex, age, musical level, and software ability. Musical level was assigned on a scale of one, no musical background, to five, professional or semi-professional performance. Musical level was determined using both an objective criterion - performance on the listening test - and subjective criteria - information about the subject's background obtained during the interview. Subjective criteria were drawn from comments such as, "I played some clarinet in the high school band" and "I never played an instrument or studied voice, but I sang in the middle school chorus and learned to read a little music." All subjects performed well on the listening test, making no more than two errors in fourteen questions (14%), with the mean at less than one error in fourteen questions.

Software level was measured from one, some programming, to five, senior architect. Table 7.9 shows the profile of subjects who participated in the study.

Training

The training stream was well understood by all subjects. Subjects rarely requested that the stream be paused, and when they did, it was to reiterate and verify a point just heard rather than to resolve a misunderstanding. The need for certain

subject	sex	age range	musical level	software level
1	male	50-59	2	4
2	male	40-49	4	5
3	male	40-49	3	5
4	female	30-39	1	3
5	female	30-39	3	3
6	female	20-29	1	4
7	male	20-29	1	3
8	male	20-29	1	3
9	male	40-49	1	4
10	male	50-59	1	5

Table 7.9: Subject profile

clarifications was uncovered during the listening part of the first few sessions. Specifically, the needed clarifications occurred in the following three places.

1. Accessor and mutator methods are represented as; two bell-like sounds at the same pitch in rapid succession.

2. Upward and downward patterns representing writers and readers, respectively, must move strictly upward or downward with possible repeated pitches, but not generally upward or downward with local reversals.

3. A method can be represented by concrete sounds, like a shopping cart, or by more abstract sounds, notably musical sounds.

Listening

The listening results for each sound stream are reported in the following subsections. Major observations are listed below:

1. 100% of the subjects could perform classification without difficulty.

2. Overall, 80% of the subjects could understand modifiers, such as an anvil mapped to *static*.

3. 50% could correctly perform specific entity identification.

4. It appeared that concrete sound associations were easier to learn and associate than abstract sound mappings.

5. All subjects could categorize object size into one of three sub-ranges. Two subjects could only do this correctly upon a second guess, after hearing all three ranges a second time.

Expenses Project Sound Stream

Subjects listened to the main sound stream for the Expenses Project, contained in the file *expenses_5sec.wav*, in which the silences between successive entities are five seconds in duration. Selected subjects who performed well were also asked to listen to portions of *expenses_2sec.wav*, containing the same sequence of entities separated by two seconds of silence.

All subjects were easily able to classify entities as packages, classes, interfaces, or methods. One subject, Subject 7, guessed twice that a method was a package, hearing something that reminded him of the satellite-like sound, but not considering that there was only one sound, not two simultaneous sounds. The same subject incorrectly identified a method as an interface, but on repetition later during the session identified the interface correctly, a possible indicator that training or usage beyond that experienced in the session might have been required. Subject 7, an upperclass undergraduate student, was the least experienced in practical software development, so he may not have been oriented toward interfaces.

On hearing a pattern of an anvil followed by an unspecified, concrete sound, occurring in the stream *expenses_5sec.wav* at 00:20, eight of the ten listeners correctly ascertained that it represented a static method, while two could not remember that an anvil signified *static*. Likewise, on hearing a pattern of an anvil followed by a bell, occurring twice at 01:48 through 02:01, several subjects could classify them as static methods and remembered that there was something notable about the bell-like sound, but could not remember that the bell signified an

accessor method.

After an initial hearing of the Expenses Project sound stream, subjects were asked to listen the the first two minutes of the stream again, notating on paper the type of each software entity, any particular characteristics of each entity they heard, and the tree structure containing the entities. All subjects were able to correctly recognize the object types, though some, including Subject 7, were previously corrected when making classification errors. Three subjects did not recognize the anvil in the same two patterns as such, indicating that a sound more clearly recognized as an anvil may be needed. All except two subjects remembered that an anvil signified *static*.

After the notation exercise, subjects were told that the first package heard in the sound stream, consisting of the satellite-like sound plus one superimposed clarinet-like sound, represented a collection of client classes, the second package (at 00:43) represented a collection of server-side classes, and the third package (at 01:28) another server-side collection. Returning to these after listening to other entities, either the first or second package was played in isolation, and the participant was asked to identify it. This task was largely unsuccessful. The subjects indicated that there were simply too many classes and methods with too little aural differentiation among them. Half of the subjects could remember and identify the package. Of these, two took several seconds to recall which was which, one of the two going through an out-loud verbal exercise to recall that the first package was that with a single clarinet tone.

Subjects had no problem classifying a method as a constructor (an example of which is heard at 02:16) or as an overloaded constructor (heard at 02:23) when heard after the initial constructor. Most subjects also had no problem classifying a finalizer (heard at 02:31), though several distinctly recalled the sound but could not associated it as a finalizer, however guessing it to be a *close* method. These were subjects with lower degrees of experience who had not had occasion to write

finalizer methods.

Subjects were told that the method at 03:00 was a factory method. On rehearing the same method later in the session, several of the professionals pointed out, without prompting, that it was a factory method, and all but one reported being amenable to the idea of the particular mechanistic type of sound heard representing a factory method or appearing in a factory class. Several subjects had to be prompted ("is there anything special about this entity?") before indicating that it was a factory method. The remainder did not recall anything special about the method. Once made aware of the sound of a factory method, the senior architects correctly extrapolated, several upon query and one unsolicited, that the class heard at 02:43 was a factory class, though this was not in the training material.

Referenced Entities Sound Stream

Subjects listened to *called_by_ExpenseDelimitedAccessStore.wav*, a sound stream consisting of entities in sequence which are called by the method *ExpenseDelimitedAccess.Store*. Some of the called entities are in the local project, in fact in the same class *ExpenseDelimitedAccess*, while others are in the external library *java.io*.

The first few subjects also listened to *called_by_ExpenseFacade_constructor.wav*, a stream of entities called by the *ExpenseFacade* constructor, all within the local project. It was determined that the information obtained from subjects listening to this stream was redundant with the other stream, and its use was discontinued.

The first entity heard in *called_by_ExpenseDelimitedAccessStore.wav* is the class *ExpenseDelimitedAccess* itself, as the methods following it are within that class. The foreground sound of the class representation is the cello note indicating *this*. By the time they heard this stream, all of the subjects had forgotten from the training stream that a single, low cello tone mapped to *this*. A few remembered that such a cello tone meant something, but could not put that something together

with *this*. It is possible that further training or reinforcement would correct the situation, but it is also possible that the cello tone is not a sufficiently intuitive mapping to *this*, and that by convention subjects are looking for a mapping to a specific class or class type rather than such a self-reference.

All subjects correctly identified the sound immediately following the class as that of an arbitrary method within the class just heard. Eight of ten subjects correctly identified the second class (at 00:14 in the stream) and its three methods as distant and therefore in an external library, as opposed to the local first class and method. The majority identified the class as a writer class, even though this idea had not been applied to classes in the training, only methods. All subjects identified the first external method (at 00:19) as a constructor and the second as a writer. Subjects had not been told that the sound of a door closing mapped to a *close* method, but a majority heard the final door closing sound and guessed upon inquiry that this would be a stream closing method. Approximately half of the subjects correctly guessed that the external class and its methods were all about stream output. In fact, the class was *java.io*. For some subjects the stream was played several times before all of these inferences were made.

Class Sizes Sound Stream

The class size stream is ordered by size, so the ordering of sound patterns played was performed manually by the investigator. The subjects all guessed the class sizes correctly. Initially, after the first sound was played, two subjects made incorrect, hesitant guesses but quickly adjusted the guess to the correct one without prompting, one immediately and one after replaying the sound pattern.

Listening Summary

Overall, the subjects were able to recognize the types of entities and their characteristics. It appears that training or experience with the sound mappings over and

above that minimally provided in the evaluation would reduce the incidence of not remembering a sound association, such as an anvil as being *static*. The subjects were easily able to infer and reconstruct the relationships expressed through aural serialization of the entities. All subjects ascertained that entities were part of or external to the project of interest. The more experienced made inferences of an structural nature, such as that a package was about managing streams without knowing its identity as *java.io*.

Table 7.10 summarizes the degree of success at recognition and understanding based on the listening exercises, amplified in some cases by participant feedback. In the table, *degree of success* is characterized as high, medium, low, or none. For characterization as a high degree of success, eight of the ten participants had to demonstrate clear and unambiguous recognition or understanding. A low degree of success was assigned when five or fewer participants demonstrated clear recognition or understanding. Anything in between is characterized as medium. *Verification* indicates whether the degree of success was determined through the listening exercise, qualitative feedback, or a combination of both.

Qualitative Feedback

Observations and ideas articulated during participant feedback were considered important when they exhibited commonality among participants or novelty even when expressed by a single participant. These observations and ideas are itemized by concept as presented below. It is noted that speculative statements such as the usefulness of the earth-air-space metaphor, while based on participants' experience, may or may not be borne out through experimentation.

Comparison to spoken text. Several subjects articulated advantages of using non-speech sound rather than spoken text. One related that speech would be "monotone", adding "it would all just blend in and I'd stop paying attention." Said another, "[It is] easier to identify a particular [non-speech]

construction/capability	**degree of success**	**verification**
intuit earth-air-space metaphor	high	listening; stated ease of learning and recalling
understand entities as successive	high	listening
understand sequences having five-second separation	high	listening
understand sequences having two-second entity separation	low	listening; not enough time to keep up with the pace at subjects' training level
differentiate package vs. class vs. interface vs. method	high	listening
identify packages, classes, interfaces, and methods	low	listening; feedback indicating difficulty and predicting failure as program size increases
Recognize language-induced modifier (static)	medium	listening; feedback that the anvil sound in use can be improved; feedback that degree of success would become high as experience increases
Recognize purpose-induced modifier (reader, writer, accessor)	high	listening
Recognize abstract modification to identifying sound based on structure, such as factory method	medium	listening. Subjects were mixed on the distorted sound used to indicate a factory method. They felt that concrete sounds would work better. They were very enthusiastic about applying this to design patterns.
Recognize concrete identifying sound, such as close method	high	listening and positive feedback
Differentiate external vs. internal entities through audio distance	high	listening and positive feedback
Recognize size ranges via drums	high	listening; feedback indicates more than three size ranges may be discernible, and the ranges can be selectable by the developer
ability to draw structural conclusions from sequences	high	feedback; the architects performed better than the novices.
receptiveness toward exploration via hover within explorer	high	feedback. Some suggested that hover or equivalent be extended to editor panes.

Table 7.10: Mapping features and their degrees of success

sound, especially [given] the way we name stuff with five names concatenated." Asked for an example, the subject articulated the four-word variable name *CustomerSalesInformationBuffer*.)

Metaphor and analogy. Most subjects believed, when asked, that the earth-air-space metaphor helped them to determine entity types. One subject indicated that any analogy, such as a door closing for a *close* method, made a mapping concept easy to understand. A subject offered an insight into her cognitive process upon hearing a sound mapping: she associated the sound with a visual image, then later remembered the visual image on hearing the sound and performs the mapping.

Difficulty of identification. A majority of subjects felt that identification of specific entities through relatively abstract sounds would be problematic when applied for programs of any size, especially for larger programs. One subject stated that the different sound mappings were necessarily "too close to one another."

Real-world usefulness. All but one subject saw usefulness in the mapping scheme and the tool. The lone dissenter, a senior architect, stated, "I could do it visually much faster." He stated that he spends a great deal of time reading others' code, raising the possibility that he is unusually fast at reading code and picking out salient features. He did feel, however, that something like reader-writer is hard to discern while reading code, so such meta-information would be of value. Another senior architect saw the value of the mapping and tool for exploration but challenged the usefulness of sound mappings "as you code." Another subject, again a senior architect, suggested that such sonification as one codes may remind us of an entity's context in ways orthogonal to the visual context. One subject indicated that it would be nice to hear, in sequence, the classes within a package in the Package Ex-

plorer to know what classes implement an interface. If she's about to change something in an interface, she can identify all of the impacted classes. Most subjects articulated the value of listening to off-screen references.

Differentiation of internal versus external entities by their audio distance was acknowledged by all participants as appropriate and useful.

Suggestions for use and improvement. Multiple subjects suggested instrumenting the editor with the sound mappings. In both the editor and the Package Explorer, either hovering over an entity or selecting the entity would play the desired sound(s). A particular expression of this idea was that sound should be placed "right in the method," because, he stated, "even in my [own] code . . . I sometimes forget what it inherits from, etc." One subject suggested extending the mapping down a level to blocks of code within a method, which would help the subject to remember "what block we're in" inside an editor window containing a great deal of text.

At least three subjects expressed enthusiasm about the possibility of extending the sound mapping to design patterns such as *Model View Presenter*[1] [22]. One subject, a mid-level developer, reported encountering Model View Presenter frequently in her work situation, so she'd hear a small subset of sounds repeatedly, reinforcing its meaning. She would use sound in the Package Explorer to "find all the Presenter classes" for maintenance purposes.

One subject expressed the possible desire to transitively hear "what called methods call." Analogously, we might want to hear *all* parents of an entity, not just the direct parent.

A subject suggested including referenced database tables in the mapping, fulfilling an answer to the question, *what tables does this entity call?* and extending sonified entities to those outside the Java realm proper.

[1] *Model View Presenter* is a frequently-used variation of the *Model View Controller* pattern [52].

Further suggestions are offered within the item below labeled "User control."

Scalability. Several subjects questioned, to varying degrees, the mapping concept's scalability to large programs. Degree ranged from doubt to being "interested to see how it scales up to a real production-level program."

Self-assessment. One subject compared his performance to his own expectations prior to the session. "I'm actually doing better at this than I thought I would." On hearing a pattern multiple times in different contexts, "I'm anticipating now. I still have a memory of this." The particular subject had been skeptical of the mapping scheme's usefulness prior to learning and using it, afterward demonstrating receptiveness to its possibilities.

Representational Accuracy. A subject observed, "We know that people don't follow naming conventions." Therefore, the subject reasoned, the process of determining the sound mapping should rely on the actual underlying semantics rather than the entity's name. like *writeData*.

User control. Several subjects thought that selecting their own sound mappings, within a mapping framework, would help them foster their own personal sound to entity associations. One subject articulated this idea as the ability to add her own sounds from a library or import them. Another, more senior subject felt that instead of ultimately having totally automated generation of the sound mapping, let the developer pick sounds within a prescribed set of constraints (like three hammer strokes instead of four for a constructor), but have pre-annotated sounds representing standard packages like *java.io*. While several subjects thought that the representation of the self-referential *this* class should be eliminated, one subject felt it should be optional, and that similar options be controllable by the developer. One option would be the number of size categories that can be heard - more than the three currently in use, but only as many as each listener would be comfortable

with. Finally, it was suggested that the amount of time between successive entities be controllable by the developer. Two seconds was considered too short, at least given the limited amount of training and experience provided in the study. Subjects indicated inability to keep up with the pace. Five seconds seemed quite adequate, and may be on the high side as developers gain experience.

Overall paradigm. Finally, one subject asked the researcher not to give up on a paradigm, not employed, of hearing simultaneous entities. "Morse code people could hear two signals, each in one ear, so class and method may be able to overlap in time."

7.3.2 Study Two Results

Trials were performed during the period November 2010 through February 2011. Twenty-four subjects, whose demographics and backgrounds are summarized in Table 7.11, participated in the experiment. In the table, *level* is either professional (prof) or student (stud). Years of professional experience are listed in the column *yrs*. Academic level is either the highest level achieved by professionals or the current level of students, where *pg* is a first-year postgraduate and *ug* is a fourth-year undergraduate. All subjects have had some exposure to Java and either Eclipse or an Eclipse-like environment. *Java* indicates whether the subject has worked regularly with the Java language (yes, no), and *Eclipse* indicates whether the subject has worked regularly in the Eclipse environment (yes, no). Musical experience can assume the values 0 (none), 1 (instrument or voice up to typical high-school level), or 2 (semi-professional level or above). Music training is also rank Subjects were largely of the same cultural background as the investigator, meaning that metaphors employed in the sonification scheme and validated by subjects cannot be assumed to be culture-independent. Several subjects were not native English speakers but have lived in the U.S. for at least five years.

The acoustical environment varied by session, as most of the sessions occurred at a location convenient to the participant. This may have caused aural recognition to vary among participants.ed 0-2, 0 meaning none, 1 some instruction, and 2 highly trained. The numeric rankings are researcher-assigned based upon responses in the participants' initial questionnaire.

Group	Level	Sex	Years	Ac. Level	Java	Eclipse	Music Experience	Music Training
1	prof.	M	25	PhD	no	no	2	2
1	prof.	M	5	PhD	yes	yes	0	0
1	prof.	M	28	PhD	yes	yes	2	1
1	stud.	M	NA	ug	yes	no	0	0
1	prof.	M	2	MS	yes	yes	0	0
1	stud.	M	NA	ug	yes	yes	0	0
2	prof.	M	30	PhD	yes	no	1	0
2	prof.	M	6	BS	yes	yes	0	0
2	prof.	M	4	MS	yes	yes	0	0
2	prof.	M	12	BS	yes	no	1	1
2	stud.	M	NA	pg	yes	yes	0	0
2	stud.	M	NA	ug	yes	yes	1	1
3	prof.	M	20	PhD	yes	yes	2	2
3	prof.	F	10	MBA	yes	yes	0	0
3	prof.	M	24	MS	yes	yes	1	1
3	stud.	F	NA	ug	yes	yes	2	2
3	stud.	M	NA	ug	yes	yes	0	0
3	stud.	M	NA	ug	yes	yes	0	0
4	prof.	M	25	PhD	yes	no	0	0
4	prof.	M	27	MS	yes	no	2	1
4	prof.	M	5	MS	yes	yes	1	1
4	prof.	F	9	BS	yes	no	0	0
4	stud.	M	NA	ug	yes	yes	0	0
4	stud.	M	NA	ug	yes	yes	1	0

Table 7.11: Subjects, ordered by group

As Table 7.11 shows, the sample contains 15 professionals and 9 students. Participant demographics are heavily weighted toward male, with 21 males and 3 females. The median professional has 12 years of experience, and the education level of professionals is weighted toward advanced degrees. Overall Java usage is high, and Eclipse experience is weighted toward regular Eclipse usage versus

occasional/past usage.

Statistical Results

The tasks CP 5 and PICT 5 are used for hypothesis testing, as they are the only multiple-step tasks of significant duration whose questions are those of the type encountered in program maintenance activities. The observed task durations, listed by treatment and ordered from lowest to highest, are indicated in Tables 7.12 and 7.13.

group	treatment	position	times	mean	median	stdev
1	sonified	first	180 210 275 445 600 624	389	360	196
2	unsonified	first	111 201 284 300 370 387	276	292	105
3	sonified	second	180 255 310 330 388 640	351	320	158.5
4	unsonified	second	135 234 283 283 285 496	286	283	118

Table 7.12: CP Task 5 times, by group

group	treatment	position	times	mean	median	stdev
1	unsonified	second	13 140 145 200 330 400	205	172.5	140
2	sonified	second	74 127 288 299 342 600	288	293.5	185.5
3	unsonified	first	112 135 159 169 180 600	226	164	185
4	sonified	first	125 140 169 179 205 327	191	174	72.5

Table 7.13: PICT Task 5 times, by group

The median, quartiles, data extremes, and possible outliers of CP Task 5 and Pict Task 5 by treatment are depicted by the box plots of Figure 7.3. Whiskers

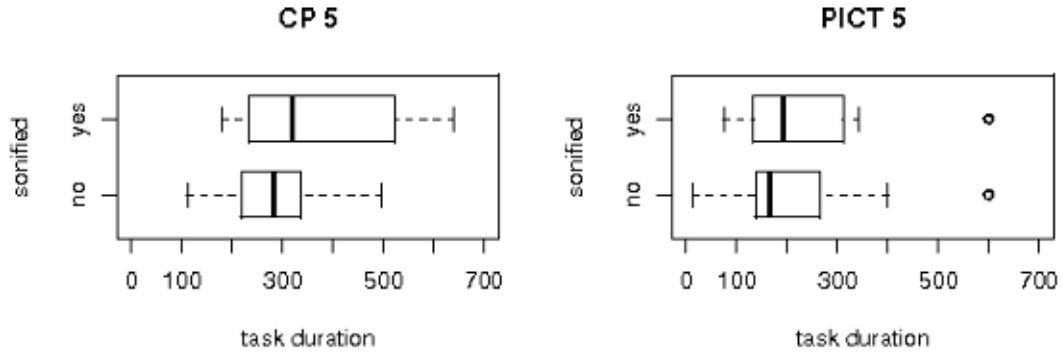

Figure 7.3: CP Task 5 and PICT Task 5 Box Plots

extend to the data extremes. For CP Task 5, the box plot for the sonified treatment consists of Groups 1 and 3. For PICT Task 5, the box plot for the sonified treatment consists of Groups 2 and 4.

Both the sonified and unsonified data for CP Task 5 or PICT Task 5 are normally distributed as determined by Kolmogorov-Smirnov tests at a 5% significance level, whose results are shown in Table 7.14.[2]

Task	Treatment	Groups	p-value
CP 5	sonified	1 & 3	0.81
CP 5	unsonified	2 & 4	0.86
PICT 5	sonified	2 & 4	0.86
PICT 5	unsonified	1 & 3	0.28

Table 7.14: Kolmogorov-Smirnov tests for normality

Sonified versus unsonified task durations were compared via the t-test. For CP Task 5, the test result t = 1.531 translates to a p-value of 0.929. For PICT Task 5, t = 0.396 translates to a p-value of 0.652. In neither case is the null hypothesis rejected at a 5% one-sided significance level. Due to the presence of possible outliers, PICT Task 5 was also evaluated using the Wilcoxon non-parametric rank-sum test. The result W = 79.5, which translates to a p-value of 0.68, similar to the t-test result.

Possible outliers, having task durations greater than the third quantile plus 1.5 times the interquartile distance as revealed by R's boxplots, are summarized in Table 7.15. There are no possible outliers with low task durations. Possible

Task	Duration	Treatment	Group
PICT 5	600	unsonified	3
PICT 5	600	sonified	2

Table 7.15: PICT Task 5 possible outliers

outliers did not arise due to experimental anomalies that can be controlled, so they are included in all statistical test results.

[2]Statistics were obtained using R, Version 2.12, and are described and further referenced in The S-Plus4 Guide to Statistics[99]. Plots for this experiment were obtained using R, Version 2.12, and are described in The S-Plus4 Programmer's Guide [100].

For each task CP 5 and PICT 5, the group performing the sonified treatment first was compared to the group performing it second to uncover possible learning effects affecting task times. Unsonified performance was similarly compared. Neither t-tests nor Wilcoxon tests support the presence of learning effects. Likewise, no difference in performance between professionals and students was uncovered.

Nine of the ten tasks yield a correctness value of *incorrect* or *correct*, determined by each subject's verbal response to the question(s) posed in the protocol. CP Task 5, whether sonified or unsonified, often resulted in an intermediate response in which the subject failed to identify one of the logging classes and proceeded to offer a somewhat weakened explanation of how to fix the logging inconsistencies. Because of the similarity of the intermediate responses to one another, an intermediate value of *partial* was added for partially correct task completion. CP Task 5 is the only task for which *partial* is employed. All responses for CP Task 5 were either partially correct or fully correct.

Task correctness when sonified was compared to correctness when unsonified using Fisher's exact test of independence. The results, reported in Tables 7.16 and 7.17, show no significant advantage of sonified over unsonified task correctness at a 5% significance level.

CP 5	Incorrect	Partial	Correct	
sonified	0	6	6	p = 0.68
unsonified	0	8	4	

Table 7.16: Fisher results for CP Task 5 correctness

PICT 5	Incorrect	Correct	
sonified	1	11	p = 1
unsonified	1	11	

Table 7.17: Fisher results for PICT Task 5 correctness

The results for each task and treatment were tested for a possible correlation between task performance and task correctness using the Kruskal-Wallis nonparametric test. No correlation was found at a 5% one-sided significance level, as

shown in Table 7.18.

task	treatment	p-value
CP 5	unsonified	0.35
CP 5	sonified	0.13
PICT 5	unsonified	0.11
PICT 5	sonified	0.11

Table 7.18: Kruskal-Wallis results

CP Task 4, although not one of the tasks used to assess the null hypothesis, is remarkable because sonified task duration in this case appears significantly lower than task duration when unsonified. The t-test yields t = -3.078, for a p-value of 0.0035. There are no outliers. No learning effect was encountered. Box plots for CP Task 4 are shown in Figure 7.4.

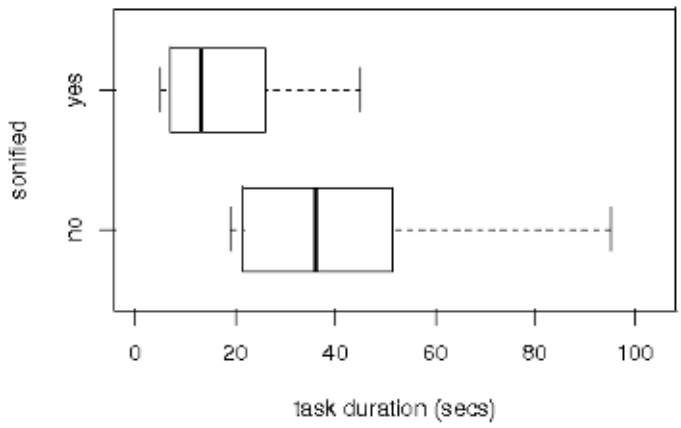

Figure 7.4: CP Task 4 box plots

The Kruskal-Wallis test reveals no correlation between task correctness and treatment, with p-value at 0.1974 for unsonified and only one group (correct) for sonified.

For tasks other than CP 4, CP 5, and PICT 5, t-tests indicate nothing remarkable, and Fisher's exact test of independence indicates no significant differences in correctness between sonified and unsonified treatments.

Interview Data

Brief recorded interviews were successfully obtained for nine of the subjects, including professionals and students spanning all four groups. Recorded information was captured after each exploration period and upon completion of CP Task 5 and PICT Task 5. Of the subjects not successfully recorded, either the recording device did not function properly, the recording device was not available, the recording was garbled, the subject provided only trivial information, or the subject wished not to be recorded. Comments from non-recorded subjects were captured in written form. Observations of all subjects were also captured in written form. Pertinent interview excerpts and observation notes are presented in Chapter 8.

Post-Trial Questionnaire

The individual session concluded with administration of the PSSUQ. The questionnaire was completed by 16 of the 24 subjects. Sessions sometimes lasted longer than the anticipated 1.5 hours. When subjects were pressed for time for that or other reasons, the PSSUQ was omitted. It was not offered as a take-home exercise because immediate impressions were desired. Box plots of PSSUQ responses are shown in Figure 7.5.

Overall, responses are in the neutral to agreement range, with fairly large spreads into the disagreement range. The overall mean of responses to the 19 questions is 3.0. The overall standard deviation is 1.6. The PSSUQ questions are reproduced in table 7.19 along with descriptive statistics supplementing those in the boxplot.

7.4 Summary

This chapter has presented the results of both an exploratory study, mainly qualitative, and a later quantitative experiment. The exploratory study demonstrated

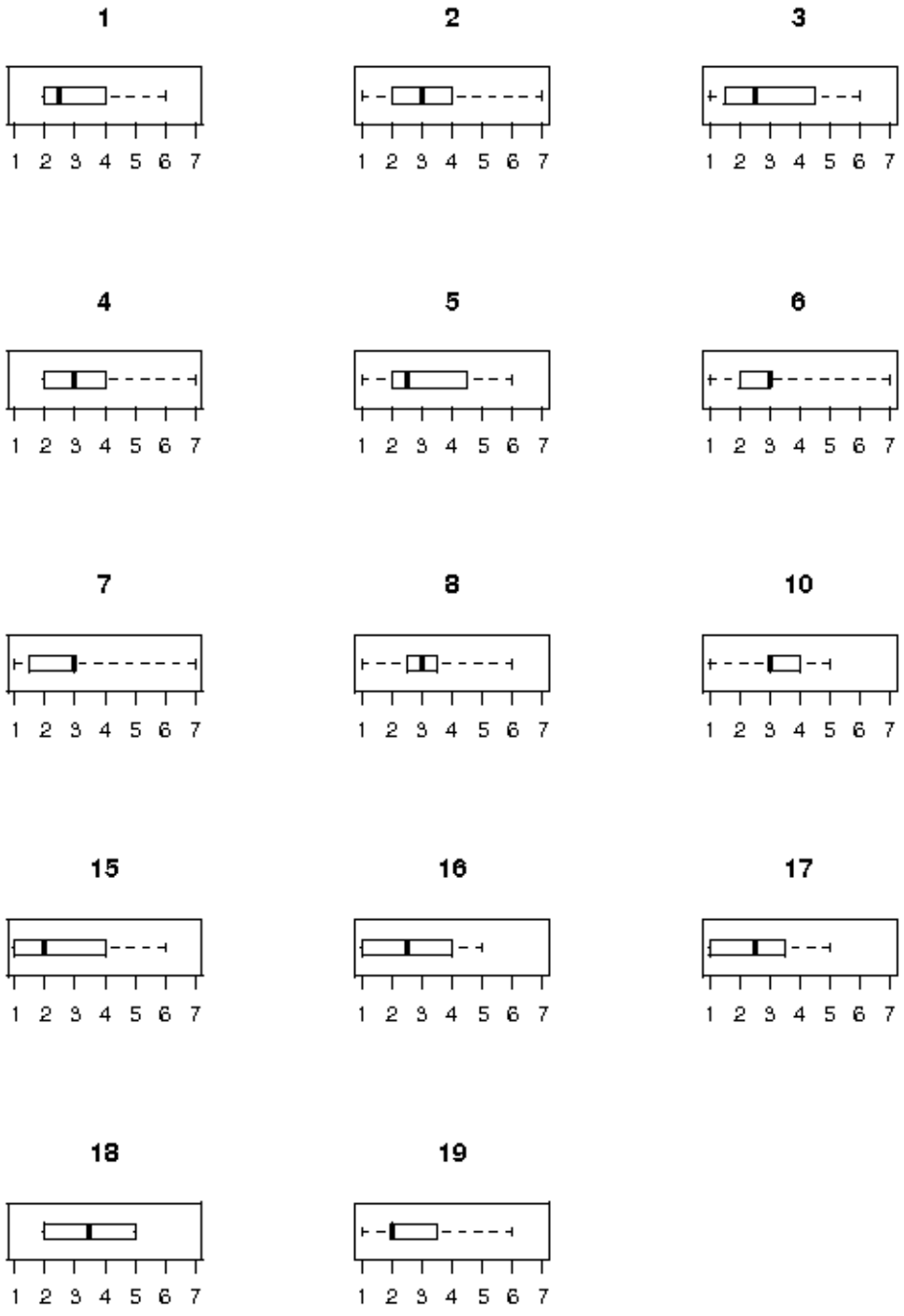

Figure 7.5: Box plots for PSSUQ questions

No.	Question	Mean	SD	Median
1	Overall, I am satisfied with how easy it is to use this system.	3.1	1.4	3.0
2	It was simple to use this system.	3.3	1.8	3.0
3	I could effectively complete the tasks and scenarios using this system.	3.0	1.8	2.5
4	I was able to complete the tasks and scenarios quickly using this system.	3.4	1.6	3.0
5	I was able to efficiently complete the tasks and scenarios using this system.	3.1	1.7	2.5
6	I felt comfortable using the system.	3.0	1.7	3.0
7	It was easy to learn to use this system.	2.9	1.8	3.0
8	I believe I could become productive quickly using this system.	3.1	1.4	3.0
10	Whenever I made a mistake using the system, I could recover easily and quickly.	3.2	1.4	3.0
15	The organization of information on the system screens was clear.	2.7	1.7	2.0
16	The interface of the system was pleasant.	2.6	1.5	2.5
17	I liked using the interface of the system.	2.5	1.5	2.5
18	This system has all the functions and capabilities I expect it to have.	3.5	1.3	3.5
19	Overall, I am satisfied with this system.	2.8	1.5	2.0

Table 7.19: PSSUQ Results

that using sound to help understand program structure is viable. The experiment's null hypothesis was not rejected; that is, sonification of program structure was not shown to improve performance in program comprehension tasks. In addition, there is no evidence that task correctness is improved through sonification. One task did lead to better performance when sonified. Chapter 8 presents an evaluation of the results.

Chapter 8

Analysis

The present chapter presents analysis based primarily upon the findings stated in Chapter 7. It was established in Chapter 7 that task performance is not of lower duration using sound than using visual search under the experimental conditions. Conversely, Chapter 7 provided evidence of situations in which task performance may be improved by use of sound, demonstrated by CP Task 4.

Section 8.1 presents analysis from a task strategy perspective based on the tasks performed in Study Two. Section 8.2 lists threats to validity for both studies. Section 8.3 extends analysis to considerations such as adoption, incorporating information from both studies. Section 8.4 presents design guidelines for entity sounds based upon the results of both studies. Section 8.5 is a brief summary of the chapter.

8.1 Exploration and Task Performance

Figure 8.1 summarizes the exploration strategies for each program in Study Two. The usual strategies for PICT were Unsonified Strategy A and Sonified Strategy A. The unsonified and sonified B strategies were employed by one subject each.

The use of sound appears to have an impact upon the way exploration is performed. Without sound, a bottom-up, linear code reading strategy is initiated,

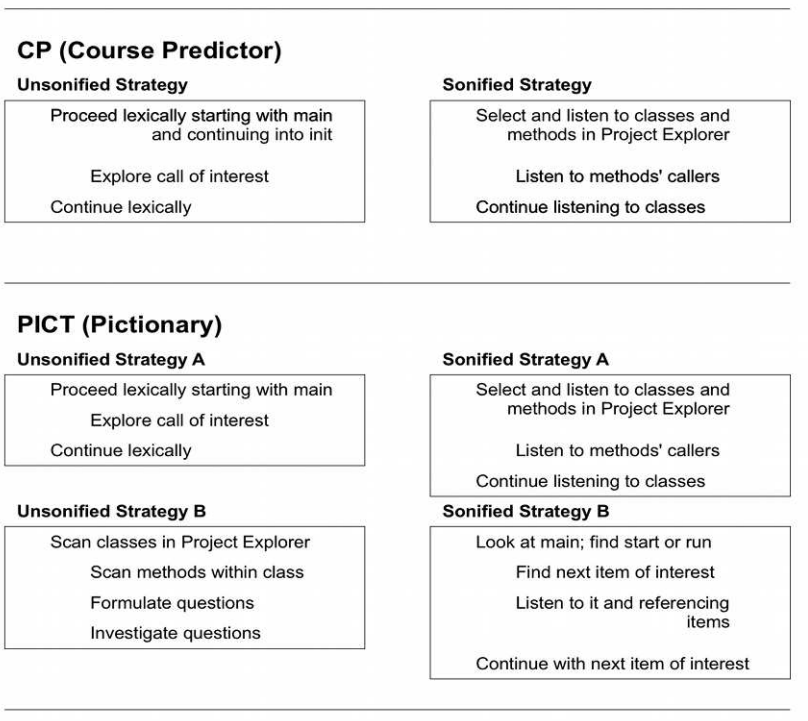

Figure 8.1: Exploration strategies

as opposed to a more breadth-first approach with sound. This may be partly because subjects are trying to memorize sounds prior to setting out on what they consider to be the actual exploration, but understanding does appear to occur during that phase and continues using a breadth-first approach.

8.1.1 CP Exploration, Unsonified

Figure 8.2 is a Unified Modeling Language (UML) class diagram [21] of salient classes and methods in the CP program, annotated to indicate the calling structure for CP Task 5.

The usual exploration strategy without sound occurred in a bottom up manner, driven by the code, with some knowledge of application behavior also used for guidance. A typical exploration for the CP program proceeded in the following manner, expanding upon the outline shown in Figure 8.1.

1. Subject locates the main method by name and identifies its class using the

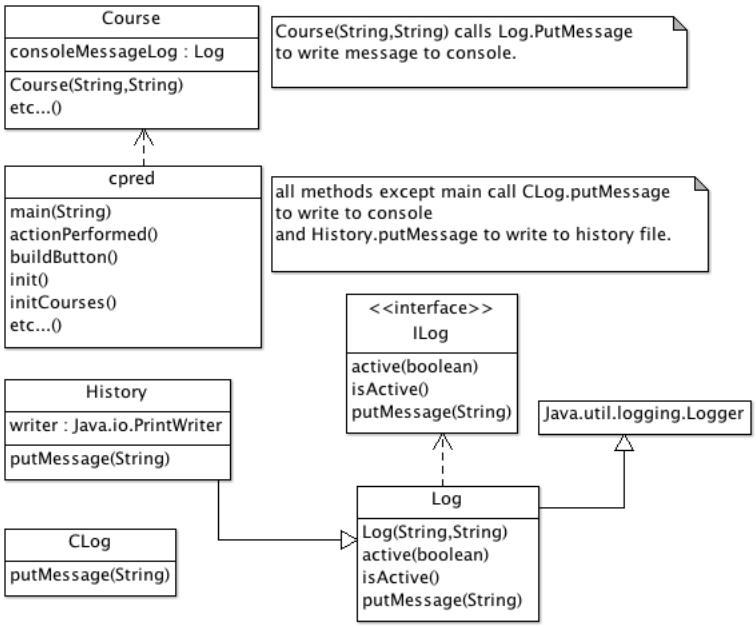

Figure 8.2: Partial CP Class Diagram

Package Explorer.

2. Subject examines the main method in an editor window. Subject notices that it is relatively brief, and that it calls *init*.

3. Subject examines *init* in an editor window, noticing that it is also brief. Some subjects notice that *init* has two logging calls. Subject concludes that the application is event driven, either by noting initialization of the action handler or by the brevity of sequential logic.

4. Subject examines *actionPerformed*, but not deeply.

5. Subject examines logging code discovered by scanning the Package Explorer or noted in the initialization code. In some cases, the subject did not reach this stage, and in other cases the subject did not opt to examine logging code.

6. Subject continues lexically and repeats drilling down.

8.1.2 CP Exploration, Sonified

The typical strategy with sound was to become familiar with some of the class and method sounds, then move on to listen to called-by and instantiated-by patterns, some leading to the visual investigation of items of interest in editor windows. Several subjects tried to memorize the sounds directly representing all of the classes and methods, leaving the remainder of the investigation incomplete.

8.1.3 CP Task 5, Unsonified

Having performed exploration, the unsonified task strategy became a search for calls to the logging methods known from exploration. Subjects often did not search for additional logging methods once several had been located, resulting in partially correct responses, in which subjects articulated two of the three logging methods *Log*, *History*, and *CLog*.

One subject initiated CP Task 5 by searching for 'log' using the File Search capability (distinct from the Java Search capability), which returns all three logging methods and the calls to each. By 45 seconds, the subject determined that 'requirements two and three are obviously not met.' Only at 135 seconds did the subject first notice *CLog* while revisiting the initial search results. The subject spent the remainder of the time clicking through all logging calls to ensure that both console and log file calls existed together each time. This subject performed one half second short of the median and two seconds above the mean, and the subject's answer was fully correct. Use of File Search and visiting all of the resulting calls appears to have assured correctness, achieved over an unremarkable task duration.

> *First was like a breadth-first search to try to hear all the different sounds that I could. I sequentially heard all the classes, then down into the methods. I didn't get time to play all the items and then to*

> *do the appropriate what class does it extend, what calls it, etc. In the editors, I was partially looking to see if this method looks interesting to me - cross-checking against the sound.*

The breadth-first nature of that subject's exploration is borne out by the observation of extensive use of sound, combined with the observation that there were six editor windows open at the end of the exploration period.

8.1.4 CP Task 5, Sonified

A subject who successfully answered the questions in relatively lengthy task duration (ranking 12^{th} of twelve) reported using the following strategy, also used by others whose correctness was full or partial:

1. The subject noticed all three logging classes visually in the Package Explorer
2. The subject played calls to all three methods, either remembering the caller's sound or else matching the caller to its sound.

A variation, used by the subject whose duration ranked third lowest and whose response was partially correct, was to determine first, by listening to the two discovered logging classes, whether either class imports an interface external to the project. *Log* does, which the subject confirmed to be the logging interface by opening the class in an editor window and very quickly observing the Class header, all within the first 38 seconds of the task. *CLog* does not, relieving the subject of the necessity to open and view it while focusing on whether it used the Java logging interface. (It would have to be opened and viewed at a different point in the task to determine that it writes to the console as its name implies, unless that had been determined during exploration.)

The subject with arguably the highest level of musical experience and training, also having a high level of professional software development experience, provided

fully correct answers and equalled the shortest duration when sonified. The subject had been able to complete the typical regimen during exploration, afterward having almost total recall of the sound associations during task performance. The subject was also able to recognize an item prior to the completion of its sound, being prepared to perform the next action immediately upon sound completion. It is notable that the other subject with the shortest duration when sonified was a student and reported having no musical experience or training. While this may have some implication, no relationship among experience, strategy, aural memory, and task duration, and task correctness was shown statistically in the study.

8.1.5 Discussion of CP Exploration and Task 5 Strategies

Exploration of CP unsonified using Java search, versus sonified, appears in general to have motivated different strategies. The typical unsonified strategy is to begin depth-first exploration of the calls in main (and then *init*). Several levels of calls are examined, though possibly in a cursory manner, then the next sequential call in the main method or the *init* method is examined. The sonified approach begins breadth-first, with the subject listening to different classes and methods to find out their sounds (knowing that tasks using those sounds were about to be performed), but also pausing to explore those of particular interest.

It may be hypothesized that the breadth-first strategy induced by using sound lends itself to the opportunistic introduction of top-down hypothesizing through early, semantically-assisted discovery of beacons representing behavior of interest. While exploring the CP program in a breadth-first manner, the subject notes, 'oh, here is our logging code, let's look a bit at that.' This strategy is not dependent on use of sound, but without the presence of sound, exploration appears to be more likely to proceed bottom-up such that code fulfilling notable functions would be discovered only after discovering it through linear reading.

8.1.6 PICT Exploration, Unsonified

Figure 8.3 is a UML class diagram of salient classes and methods in the PICT server program, annotated to indicate the calling structure for PICT Task 5.

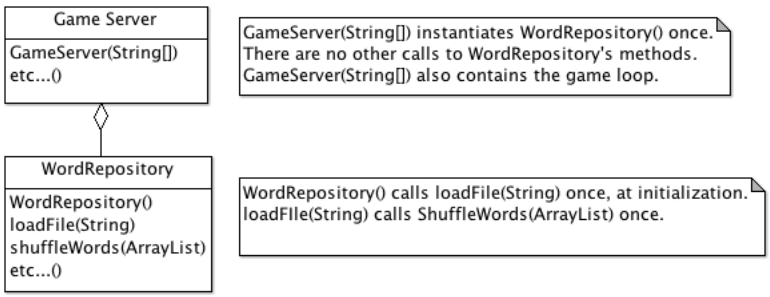

Figure 8.3: Partial PICT Class Diagram

Unsonified exploration of the PICT program was performed using one of two strategies, Unsonified Strategy A and Unsonified Strategy B for PICT in Figure 8.1.

Unsonified Strategy A for PICT is similar to Unsonified Strategy A for CP, to proceed lexically through main and explore calls of interest. Unsonified Strategy B was employed by a particular subject, an experienced software engineer, who scanned through the classes in editor windows to 'see where the bulk of the work is, try to get a flavor of what the methods are, what the class variables are, what does it extend, eyeball the structure.' The subject proceeded at least somewhat top-down by using domain knowledge to pose hypotheses or answer self-constructed questions. One of the questions addressed by the subject during exploration was, 'how does it do its randomization?'

It appears that it was possible to learn almost the entire program during exploration, although only a few of the subjects appeared to accomplish that due to the time limit.

8.1.7 PICT Exploration, Sonified

The usual strategy, PICT Sonified Strategy A, was to begin by listening to the sounds of various items, then learn something of the structure by listening to the methods' callers and class instantiators. Subjects skimmed the sequential logic in the *main* and *run* methods, and most also skimmed *WordRepository*. A subject (experienced software professional) whose ensuing task duration ranked at the median described exploration:

> *Strategy was to explore the three classes inside the package. I could see the structure of the program a bit from the three names, main and server and WordRepository, so it was a matter of figuring out who was calling whom. I ended up using the sounds to verify assumptions that the main calls server, that server calls word repository, which would be classic structure for this. I discovered interfaces and a couple of things with sounds. [Didn't look at so much code.] For the structure, I would be looking at method headers, and a lot of the method headers are in the [Package Explorer], so if we have a type 'list' and there's a method 'getList,' then I could deduce a lot of structure from that.*

One subject in particular dispensed with playing various items to learn their sounds, generating Sonified Strategy B. This subject immediately set out by investigating the main method and its calls selectively, resulting in an exploration well-suited for the ensuing PICT Task 5:

> *I used the same method I did before by going to main first, and I didn't use sound until I went to 'start.' I went to look at start. I went to the run method - first I looked at the way game server was created, and I went to the run method. Then I wanted to know about getNextWord, so I used sound for that, and I looked at where it comes from - what*

> *calls it, instantiates, for example wordRepository. I wanted to know where it was instantiated.*

8.1.8 PICT Task 5, Unsonified

The essential strategy for performing the task was to either remember *shuffleWords* or select it based on its name, search to see where it was used, determine its caller hierarchy, and note that it is only invoked upon initialization.

Performing the subsequent task became a simple matter of recall. The strategy is one that the subject had gravitated toward years ago while understanding programs and still employs routinely. The subject completed PICT Task 5 in 13 seconds, ranking lowest in duration, over two minutes shorter than the median and over three minutes shorter than the mean.

Other subjects proceeded by using the main method as a starting point and exploring sequentially, pausing on calls of interest to explore them more deeply, thereby using the bottom-up comprehension strategy often observed during task CP 5 unsonified. One such subject, an experienced software engineer, also relied on knowledge gained during exploration, specifically the existence of *shuffleWords* and its call hierarchy. According to the subject, 'it was a matter of looking through those loops and keeping track of the number of times it was calling shuffle." The task required 112 seconds, the second lowest duration. A student, using an almost identical strategy, fared almost as well, taking 159 seconds and ranking just beneath the median.

8.1.9 PICT Task 5, Sonified

All subjects used variations of a single strategy for performing the task. This may be due to the smaller size of PICT relative to CP. A canonical sequence of steps follows:

1. Subjects recalled from exploration that the method *shuffleWords* performs

the randomization, or they gleaned that as a first task step by seeing its name in the Package Explorer.

2. Listen to the caller(s) of *shuffleWords*, which turns out to be *loadFile*. At this juncture, some subjects recalled the association of *loadFile* with its orchestral-excerpt sound, some did not. Of those who did not, some could identify its class via its sound, most could not. The ability to associate these sounds and items appear to have impacted the following step's duration.

3. Those who could not recall *loadFile's* sound association searched for *loadFile*, sometimes by sequentially searching all methods, sometimes via some heuristic or random search strategy. The subject who provided an incorrect response and consumed 600 seconds failed to locate *loadFile*. Subjects who had trouble locating *loadFile* had to occasionally replay the caller of *shuffleWords* to refresh the sound in their mind.

4. Once *loadFile* had been located, the subjects played its caller(s), determining within a few seconds that the sole caller is *WordRepository's* constructor. The only other constructor is that within *GameServer*; playing the sound of each constructor's class clarified which constructor was indicated.

5. Some subjects opened *WordRepository* to learn or help themselves recall that it calls *loadFile* once upon instantiation.

6. Subjects played the caller(s) of the *WordRepository* constructor, or they played the instantiator(s) of the *WordRepository* class, to find that the *GameServer* constructor is the sole caller.

7. Subjects searched or scrolled within *GameServer* to find that *WordRepository* is only instantiated upon initialization.

A variation was to guess *loadFile* in the first step, look at it, and determine that *shuffleWords* is where the actual randomization occurs. Another variation, used

by several subjects including the subject whose task duration was shortest, was to look listen to caller(s) of the method *getNextWord*. Subjects reported immediately recalling the car-starting sound as being associated with the *run* method. Those that did not recall *run* specifically recalled that the car-starting sound was associated with some sort of initialization or startup, and quickly deduced the correct method.

Ten of the twelve subjects performed this task/treatment combination at durations within a range of 125 seconds to 342 seconds. Five of the ten performed in the narrower 125 to 179 second range. The narrower range, compared to task CP 5, is in accordance with the lesser size of PICT 5 in terms of both number of methods and lines of code (see Figure 7.3 and Table 7.5).

One subject performed PICT Task 5, in 74 seconds, the shortest duration among subjects performing the sonified treatment and second shortest for both PICT Task 5 treatments. Having had no particular musical experience or training, this subject was able to utilize sound selectively without a familiarization period.

Subjects not only reported immediate association of the car-starting sound, but most also reported the ability to immediately recall the hammering sound as that of a constructor. Both sounds are of the concrete variety, suggesting that concrete sounds are easier to associate. The hammering to constructor sound association had also been well rehearsed. The car-starting sound had been encountered a few times during exploration, so it was not as well-rehearsed as the hammering sound, but it had been learned within a half hour of performing the task.

8.1.10 Discussion of PICT Exploration and Task 5 Strategies

For this task, examining what turn out to be the the right entities during exploration appears to have shortened task duration. The chances of that happening for the PICT server are better than for CP because the former's code base is

smaller. Beyond that, the best-performing subject was either fortunate to have paid particular attention to the logging code or correctly anticipated what might be asked when performing the tasks. Memory and experience may play into this just as in CP Task 5. CP. The task itself was then reduced to discovering or verifying the call sequence to do the randomization and noting where call origination occurs relative to the progress of the game. In this case, the sonified and unsonified strategies were equivalent.

8.1.11 CP Task 4, Unsonified and Sonified

CP Task 4 is the single task out of ten tasks in which sonified task performance was lower, overall, than unsonified task performance using the Java Search feature. For CP Task 4, the subject is asked to determine the callers, if any, of the method *URLButton.actionPerformed*. A successful unsonified strategy is to perform a search for references to *actionPerformed*, finding no callers internal to the project, then either recall or intuit from its name that *actionPerformed* is an event handler or look at its class header to see that is an event handler. A short cut was to initially guess by its name that *actionPerformed* is an event handler, and further guess that no code internal to the project would call it. No subject reported employing the short cut. The successful sonified strategy is to listen to the caller(s) of *actionPerformed*, hearing 'infrastructure' outside the project as the single result. Another successful short cut, for use with either treatment, is to have determined the answer during exploration and recall it, but again, no subject reported being so fortunate. The strategies articulated above will be referred to as baseline strategies.

Lacking detailed task strategy data from interviews, the unsonified and sonified performance of CP Task 4 are compared to theoretical minima. A keystroke-level analysis, using the Keystroke-Level Model (KLM) [31], provides an idea of the minimum duration for the unsonified and sonified baseline strategies to reach the

correct conclusion. KLM assigns standard times to keystroke and mouse actions performed by the application user, in this case the subject. The sum of those times is the minimum time necessary to perform the complete task. Thinking times are not included, except as noted. Possible error paths are also not included. KLM Operators and standardized times, except for mouse positioning (P), are those given by Kieras [79]. Positioning time is reduced from Kieras due to proximity of items. Average Positioning (P), Wait times (W) for system responses, and think times (T) for reading have been measured on a single-user Apple Mac Mini desktop system which was also used by experimental subjects. The KLM analysis offers an estimate of minimum task duration for an average user. A minimum task duration computed using KLM can be exceeded on a keystroke level by an above-average user. Overall, relative think times, reflecting relative cognitive load, can be estimated by measuring actual task times against KLM's theoretical task times.

The initial state of the environment is shown in Figure 8.4.

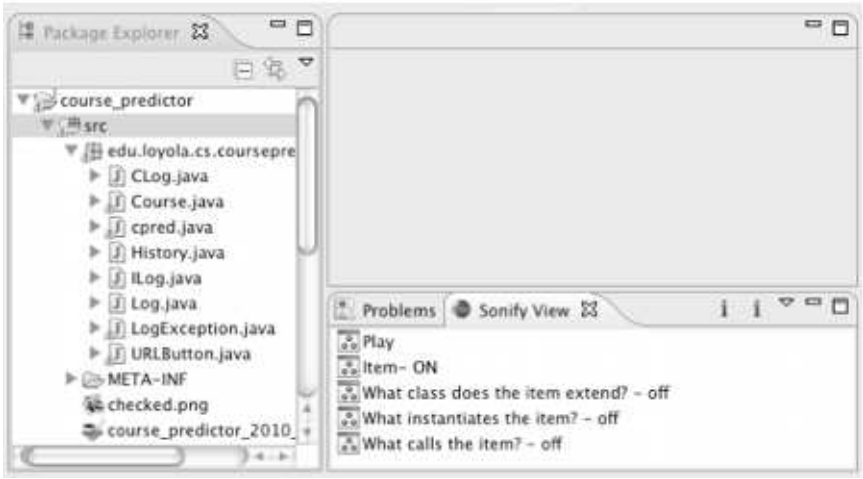

Figure 8.4: Initial state of environment for CP Task 4

Analysis of the baseline task duration for the unsonified treatment is shown in Table 8.1.

Analysis of baseline task duration for the sonified treatment is shown in Table 8.2.

step	time (sec.)	operator	description
1	0.0		Initial state: URLButton.java visible in explorer, hand homed to mouse
2	0.7	P	Move mouse to URLButton arrow icon
3	0.2	BB	Click URLButton arrow icon (to expand it)
4	0.7	P	Move mouse to actionPerformed
5	0.1	B	Right-click actionPerformed, holding down mouse button (to view a context menu)
6	0.2	W	Wait for context menu to appear
7	0.7	P	Move mouse to References in context menu
8	0.2	W	Wait for second-level context menu to appear
9	0.7	P	Move mouse to Workspace or Project in second-level context menu
10	0.1	B	Release mouse
11	0.2	W	Wait for search results tab to show no callers
12	1.0	T	Read search results
11	0.7	P	Move mouse to actionPerformed
12	0.4	BBx2	Double click actionPerformed (to view it in an editor window)
13	0.5	W	Wait for actionPerformed to appear in editor window
14	0.5	T	Read header
	6.9		TOTAL - BASELINE MINIMUM DURATION

Table 8.1: KLM analysis of CP Task 4, unsonified

For this task, the single required sound consumes 3.4 seconds, and additional silence is added to ensure there is no more than one caller, rounding the sound experience to 4.0 seconds. While listening, the astute subject recognized the infrastructure sound, or at least that it represents something external to the program, and realizes that it is the only caller. Similarly, for the unsonified treatment, the subject reads the method header and realizes that it an event handler. One must deal with the relatively lengthy (in terms of KLM operators) time for playing the sound, but has less steps to perform compared with the unsonified treatment. The KLM-estimated unsonified baseline task duration is 1.1 seconds less than the sonified task duration.

KLM estimates versus actual task durations for CP Task 4 are shown in Table 8.3. All durations are in seconds.

step	time (sec)	KLM operator	description
1	0.0		Initial state: URLButton.java visible in explorer, hand homed to mouse
2	0.7	P	Move mouse to URLButton arrow icon
3	0.2	BB	Click URLButton arrow icon (to expand it)
4	0.7	P	Move mouse to actionPerformed
5	0.2	BB	Click actionPerformed
6	0.7	P	Move to 'what calls this item?' button
7	0.4	BBx2	Double click button
8	0.7	P	Move to 'Play'
9	0.4	BBx2	Double click Play
10	4.0	T	Listen to sound; ensure it is the only sound (i.e., the only caller)
	8.0	W	TOTAL - BASELINE MINIMUM DURATION

Table 8.2: KLM analysis of CP Task 4, sonified

Treatment	KLM	min	q1	median	mean
Unsonified	6.9	19	21.75	36.0	41.9
Sonified	8.0	5.0	7.5	13.0	17.0

Table 8.3: KLM estimates vs. actual task durations

Two subjects performed the task in less time than the KLM estimate, and three subjects equalled the estimate. Possible explanations include significant recall from exploration, guessing, and faster mouse movement and wait times than estimated by KLM. However, these explanations can also apply to the unsonified treatment, for which no subject came close to the KLM estimate. Treatment-specific explanations include not having to listen to the entire sound because the subject is merely affirming the probable answer, not having to listen to the entire sound because the subject can recognize 'infrastructure' within the first second of playing the sound. Subtracting three seconds of the sound brings the KLM estimate to 5 seconds. Even given that consideration, the data suggest that either more cognitive load is encountered or repeating of actions is done by subjects given the unsonified treatment.

It is notable that no subject given the sonified treatment incorrectly reported the answer. While the previous chapter indicated that no correctness advantage

is inferred on a statistical basis, there is a logical basis for the 100% degree of correctness using sound. Subjects not using sound could not necessarily conclude that anything at all calls *actionPerformed*, because there is no information on anything external to the program. Subjects using sound are given positive feedback that the something within the system infrastructure (the event controller in this case) does call the method. Certainly calls of external origin can be added to displayed information, but in reality such information is not included in the Eclipse environment. It is possible that the positive feedback made subjects certain of their answers more quickly than those performing the task unsonified. Finally, it is notable that the first-quartile subjects using sound are a mixture of professionals and students at all levels of musical experience and training.

Discovery of external entities is limited to callers and instantiators in the experiment, which uses a restricted set of sound associations compared to those projected for the ultimate production version of the tool. Specifically, what an entity calls, including calling targets outside the program code proper, is omitted. Possible tasks on the basis of that capability have not been considered. Another restriction is that of sound to the Package Explorer. The envisioned production tool would provide the ability to select and listen to entities within editor windows, inline in the code. Subjects' suggestions during the first study pointed out the added usefulness of the missing feature.

The length of some of the sound realizations and the inability to cut off sounds when desired may have had an impact on the experiment by lengthening task performance times given the sonified treatment. Two examples of how cutting off aural entity sequences and individual entities follow.

- The listener may desire to know whether *anything* calls a selected method. As soon as the listener hears any entity, the remainder of the sequence need not be heard.

- The listener desires only to know that an selected entity is static. As soon

as the static sound (an anvil stroke) has been heard, the remainder of the entity's sound need not be heard.

8.1.12 Concrete versus abstract sounds

Concrete sounds such as the car-starting sound of a *run* method and the hammering sound of a constructor appear to have facilitated task performance more effectively than abstract sound associations with limited rehearsal. Some search for a sound's association was often necessary when it is encountered. Even when search was not required, a several-second pause was often needed to remember the association. The strong association of a concrete sound both the need to search and the pause, corroborating Mustonen's observations and classification [112].

Dingler's conclusion that earcons are much more difficult to learn than speech [41] supports the suspicion that learning has not fully occurred during Study Two in the given time frame. A longer-term learning regimen than that offered in Study Two may impact task performance and correctness by improving the speed or ease of their associations to software entities. Dingler's study only informs short-term learning.

8.2 Threats to Validity

This section lists threats to validity for both Study I and Study II.

Study	**Type**	**Title**	**Threat Description**
I	internal validity	Saturation	It cannot be certain that saturation in terms of new qualitative information has occurred, especially given a study with a small number of participants.

Study	**Type**	**Title**	**Threat Description**
I	external validity	Single Program	Sound associations were tested in Study I against a single, relatively small Java program, which may not be representative of other programs.
I, II	external validity	Possible Subject Selection Bias	Professional software developers and some students were recruited personally by the researchers and their associates. It is possible for that reason that the sample is not a representative cross-section of the software development community.
I, II	external validity	Cultural Bias	While diversity was sought, the subjects are primarily Caucasian, predominately male, and all residents of the United States. This may affect the association of sounds, especially those of the concrete variety whose meanings may be biased by culture or gender.
II	construct validity	Task Duration	Measured task duration may not accurately reflect time to solution. Duration is measured to the point at which the subject declares arrival at a solution. Some subjects may be confident of a solution as soon as it occurs, while others may reflect on it before declaring it. While this likely does not impact the overall outcome, it may impact the comparative analysis of individual trials.

Study	**Type**	**Title**	**Threat Description**
II	external validity	Strategies	Students' comprehension strategies may differ from that of professionals, and professionals' strategies may differ by age and background. The overall statistical result measures a mix of students and professionals and therefore may not be applicable to the professional community as a whole.
II	external validity	Program Selection	Program comprehension tasks were measured in Study II against two Java programs of similar size, both prepared for production but in an academic environment. The tasks against these programs may not be representative of tasks against programs of other size, scope, language, domain, etc.
II	external validity	Task Selection	The measured tasks are of relatively short duration and are possibly not representative of the most common program comprehension tasks found in industry.
II	external validity	Single Sound Scheme	Outcomes may vary given alternate sound schemes, the ability of the subject to select a sound scheme, or the partial ability of the subject to design his or her own scheme.

Study	Type	Title	Threat Description
II	internal and external validity	Training	The level of training for each study was limited by practical constraints. Study II's task durations and observed comprehension levels may be impacted by the limited training and limited pre-trial tool usage. In industry use, longer training time and extended tool use may result in lower task times or higher comprehension levels.
II	internal validity	Locations	While consistency between the two locations was sought, there may be unobserved variation in administration or measurement between the two geographic locations, given different examiners.

Table 8.4: Threats to validity

8.3 Other Considerations

The PSSUQ survey and adoption of the concept and tool are considered in this section.

8.3.1 PSSUQ Survey

The overall results of the PSSUQ responses suggest that the tool is usable without difficulty. The results also suggest that the tool is in need of improvement which may facilitate future task performance. Table 7.19 in Chapter 7 listed the results of responses to fourteen PSSUQ questions pertaining to the sonification tool, collected from subjects during the task-based experiment. Table 8.5 pro-

vides summary statistics over the fourteen questions. The grand mean is very

low mean	high mean	grand mean	low stdev	high stdev	avg stdev
2.7	3.5	3.01	1.3	1.8	1.58

Table 8.5: PSSUQ summary statistics

close to the center value of the five-point Likert scale, and the lowest and highest means are both within a half point of the grand mean. The arithmetic average of the standard deviations indicates a broad spread. The responses to each question covered a wide range, none clustering at either extreme. It is possible that one or more of the following are true:

1. the subjects as a group did not have strong positive or negative opinions of the tool's usability,

2. the subjects as a group took into consideration that the tool is a prototype, which may mitigate negative perceptions,

3. the features provided by the tool are too few for usability to be adequately captured by the PSSUQ.

Insight may be provided by the low and high means. The low mean, 2.7, is over the responses to 'the organization of information on the system screens was clear.' This suggests that the Sonification View is in need of improvement. It is clear that the Play button can be eliminated or better differentiated from the selection buttons. It is also clear that functions can be combined such that no selection button need to apply only to classes or only to methods. It was suggested by one subject that additional audiovisual aids may serve to facilitate memory of sounds during the learning phase or when unfamiliar, rarely-heard entities are encountered aurally - a 'sound search' feature. The high mean is over the responses to 'this system has all the functions and capabilities I expect it to have.' As the subjects are newcomers to the sonification concept, it would seem likely that most would not have expectations beyond those they actually encountered.

8.3.2 Adoption Issues

There is not yet a compelling case for adoption of the sonification concept or tool presented herein. The reasons are threefold:

1. The studies performed in this thesis did not indicate significant task performance improvements.

2. Much of the qualitative feedback from the first study is positive about trying out the sonification tool during actual maintenance work. On the other hand, statements by some experienced software engineers during the second study about using the same comprehension strategies for a number of years suggest that adoption may be difficult due to reluctance or inability to shift strategy when necessary.

3. Many developers listen to music using while programming. To facilitate adoption, a tool may have to allow listening to music but interrupt the music with sound on demand.

Further evolution of the tool and the scope of the sound mapping, along with further experiments and case studies involving a wider set of tasks, may help to alter the outlook for adoption. However, diffusion of technology is a complex process that, when successful, can be lengthy. Rogers [132] expresses diffusion as a form of social change stimulated by effective utilization of communication channels. Rogers indicates that diffusion can happen in either planned or spontaneous ways, possibly involve multiple interrelated innovations, and require the assistance of change agents and opinion leaders. According to Rogers, diffusion progresses according to an S-shaped curve that enters its largely vertical component only after early adoption by 10 to 20 percent of the candidate community. Redwine and Riddle [129] found, through analysis of case studies as of 1985, that maturation times for technologies (including methodologies and conceptual constructs such as abstract data types) took 11 to 23 years before reaching the point at which

widespread diffusion was possible, requiring six major phases from basic research through propagation to 70 percent of the candidate community. Pfleeger [124] surveyed technology adoption literature, highlighting successful practices. Pfleeger brings to the fore the role of gatekeepers, who evaluate emerging technologies to determine if they address organizational needs, exhibit cost-effectiveness, and meet a variety of other criteria. Pleeger also points out that stakeholders may be promoters or inhibitors of new technologies.

8.4 Guidelines

Based on the reference mapping scheme and the information garnered through its validation, a set of rules for best determining the mapping of entities to their foreground sounds is set forth. The rules below are listed in precedence order from highest to lowest.

1. The core entity foreground sound pattern should be between 0.5 seconds and 3 seconds in length. **Rationale:** This was the span in the study, and the subjects were able to process sound patterns within this range.

2. If a concrete audio representation is available and usable, select one. Examples: a door closing for a *close* method, a shopping cart for a shopping cart class. Constraints: should not be used if the entity has a structural modifier, that is, its foreground sound may change based on its place in the architecture. A *writer*, which is characterized by an upward musical pattern, cannot be represented by a concrete auditory representation. If the entity belongs to a collection of similar entities by function, the audio representation should also belong to a class of such representations. For example, *close* methods should always be represented by door closing sounds. **Rationale:** concrete sounds which subjects can easily identify proved fastest and most accurate for recognition of entity characteristics.

3. If a concrete audio representation is not available, use an abstract or musical representation. The timbre (sound quality), pitch range, and rhythm can carry architectural and identifying information. For example, a series of similar, harsh timbres might indicate a factory class or method. In the reference mapping, pitches moving strictly upward indicate writing or posting ('putting up to'), while strict downward patterns represent reading or getting ('taking down from'). **Rationale:** subject performance was good with these representations, and these types of sound patterns can carry significant information and are flexible.

4. For entities with commonly recurring functionality, the functionality is more important than the entity's specific identification. In the reference mapping, *get* and *put* methods are all bell-like sounds. The pitch of the bell can vary while remaining within a generally treble range. **Rationale:** Sometimes identification of these entities is near-trivial, architectural information is better understood in this sonification scheme, and architectural rather than identifying information may reduce visual context switches.

5. Similarly, for entities with a known role in a design pattern, that role is more important than specific identification. Again, a factory class is an example. **Rationale:** Same as previous.

8.5 Summary

This chapter has presented an evaluation of the results from both studies described in Chapter 7. It has also presented a set of design guidelines for mapping sounds to software entities, generalizing the reference sound mapping.

Chapter 9

Conclusions

9.1 Introduction

This chapter summarizes the contributions of the thesis and articulates conclusions in light of the research question and success criteria. An agenda for future work is set forth.

The remainder of this introduction summarizes the contributions of this thesis. Section 9.2 addresses the success criteria stated in Chapter 1. Section 9.3 addresses the research question, propositions, and associated conclusions. Section 9.4 describes an agenda for future work. Section 9.5 concludes the chapter and the thesis.

As in previous chapters, the term *developer* indicates the individual using Eclipse and its sonification extension.

9.1.1 Contribution

This thesis advances the state of research into the use of sound in program comprehension.

Due to visual clutter, the need for visual context changes, and under-employment of the human audio channel, sonification, or the representational use of non-speech

sound, is seen to be able to supplement visual means of understanding the static structure of programs. The following research question was formulated and expressed in Section 1.3:

> *Can sonification supplement visual information to support comprehension of the low-level, static structure of non-trivial computer programs?*

It was postulated that a mapping of sound patterns to the low-level structural entities of a computer program, e.g. the packages, classes, interfaces, and methods in a Java program, can facilitate a maintainer's understanding of the program. A set of propositions and success criteria were established in Section 1.3. The propositions stated that a viable sound mapping, incorporated into an integrated development environment (IDE), can help to identify software entities, characterize them, and characterize their relationships, improving program comprehension performance.

A sound mapping was devised and incorporated into a prototype demonstrator tool. The sound mapping is described in Chapter 4. In addition to entity identification, entity characterization, and relationship characterization, the mapping includes aural representation of a size metric, the number of encapsulated methods in a class. The tool, an extension to the Eclipse IDE with Csound as a back-end sound generation engine, is described in Chapter 3.1. It enables the developer to select an entity and listen to that entity's referencing entities or the entity itself. An entity's parents, such as as method's class and package, and also its referencing entities, are heard as a sequence of their respective sounds. An entity itself is represented as the layering of an underlying sound (for packages and classes), an identifier sound, and modifiers that indicate either language-specific indications such as a Java method being static, or semantic indications such as a method's role as an accessor.

Two studies were performed. Study One was a human-subjects study to informally verify that the mapping is viable, that is, easily learned, understood, and

retained in the short term. The design of Study One is described in Section 7.2.1. Ten software professionals participated in individual sessions consisting of training, listening, and interviews. Training was provided in the form of a fourteen-minute, pre-recorded audio stream explaining and demonstrating the sound mapping. Listening consisted of interpreting information from five sound streams drawn from application of the sound mapping to a small Java expense-recording program. The interview provided a vehicle to discuss the subject's experience and elicit impressions and improvement suggestions. Study Two was a 24 subject, 2 x 2 crossover experiment to determine whether using the tool improves the performance of program comprehension tasks. The design of Study Two is described in Section 7.2.2. Subjects were software professionals and advanced computer science students. Each subject was provided advance web-based training, followed by participation in an individual session. Each session commenced with additional, in-person training, following which the subject performed program comprehension tasks on two production-quality programs. Each subject performed half of the tasks with a sonified treatment and the other half unsonified. Subjects were observed during initial exploration of each program and subsequent performance of the tasks. Subjects were also interviewed after their explorations and completion of the final task for each treatment. The interviews were aimed at eliciting the subjects' task strategies.

The results of Study One are reported in Section 7.3.1. The results indicate that the sound mapping can be understood and retained over the course of the session. The study suggests that characterization of entities and their relationships was stronger than identification of specific entities.

The results of Study Two are reported in Section 7.3.2. The results indicate, at a 5% significance level, no improvement (decrease) in task duration when using sound. The results also suggest no advantage in terms of accuracy of understanding. The result do, however, suggest lower task duration may be achievable given

the sonified treatment under some circumstances.

Chapter 8 provides analysis of the results of the studies. The analysis indicates that, in some cases, a sonified treatment motivates a different task strategy than does an unsonified treatment. The sonified strategy is mainly breadth-first, the unsonified strategy mainly depth-first. Both strategies primarily utilize bottom-up comprehension strategies informed by top-down domain knowledge. Given this finding plus the exceptional finding when listing a method's callers and the limitations of the study, it is possible that future research will uncover and categorize tasks in the comprehension of static software structure for which sonification is advantageous. Section 8.4 contains a set of generalized guidelines for creating sound mappings to software entities.

9.2 Criteria for Success

Section 1.4 lists five criteria for success, all of which have been met by constructing the reference sound mapping, developing the tool, and performing two studies.

1. *Define a reference sound mapping to static software entities for one programming language.* A reference sound mapping to static software entities was devised. The mapping associates earcons and auditory icons to the packages, classes, interfaces, and methods in Java software programs. The mapping consists of overlapping and sequentially-presented sound patterns, both concrete (such as a door closing) and abstract (often musical motives). The mapping makes use of recorded sounds, sophisticated synthesis and processing techniques, and localization to provide a rich aural experience. The mapping was realized for three programs: an in-progress expense reporting program, a course predictor program, and the server part of a word-guessing program. The mapping is fully described in Section 4.1.

2. *Evaluate the reference sound mapping concept.* Study One, described in Section 7.2.1, provided informal validation that sounds and sequences of sounds in the mapping can be readily understood and retained. The results of Study One, reported in Section 7.3.1, indicate that the categorization and characterization of software entities by listening to their sound representations is viable. Identification of specific entities is less successful. Table 7.10 summarizes participants' success in recognizing the audio constructs. Study Two, as analyzed in Section 8.1, strengthened and supplemented the impressions gained via Study One. Concrete sounds' associations are easier to recall than abstract sounds, involving less cognitive overhead. The identification of specific entities meets with limited success because of the problem of representing the relatively abstract software entities with sound combined with the large number of sounds necessary.

3. *Apply the mapping to program comprehension tasks.* Study Two, an experiment described in Section 7.2.2, demonstrated that the comprehension of static program structure is possible using sound as an aid. However, as indicated in Section 7.3.2, no significant advantage was shown in either task duration, the primary consideration of the experiment, or task correctness, a secondary consideration. Prior to performing any tasks, each subject was allowed an exploration period with either the sonified or unsonified treatment. It was shown that exploration of one of the programs unsonified using a Java search facility, versus sonified exploration, appears to have motivated different task strategies, as discussed in Subsection 8.1.5.

   Although only the two ultimate tasks were used to test the hypothesis, all task durations were recorded, and one of the penultimate, short-duration tasks suggests that circumstances involving off-screen entity relationships may reduce task duration under a sonified treatment. The results for that task are reported in Section 7.3.2 and discussed in subsection 8.1.11.

4. *Develop a prototype demonstrator tool using an instance of the reference sound mapping.* A tool was developed and subsequently utilized in Study Two. Described in Section 5.2, the tool consists of an extension to the Eclipse IDE integrated with Csound, a high-end sound synthesis and processing engine. The Eclipse extension includes a visual component that enables the developer to select which kind of sound association to hear and play it. Entities to hear are selected in the Eclipse Package Explorer.

9.3 Propositions and Conclusions

Chapter 1, Section 1.3 stated an overall research question and five propositions. The research question asked if sonification can supplement visual information to support comprehension of the low-level, static structure of non-trivial computer programs. Results from the task-oriented experiment, using two relatively small (by industry standards) but non-trivial programs, supports the notion that non-speech sound is usable for program comprehension when the primary activity is source code reading and search, but it does not convincingly support the notion that non-speech sound can add value in terms of increased program understanding or decreased task time. Conclusions based on each of the five propositions are presented in Table 9.1.

9.4 Future Work

Future research that centers around three areas is envisaged: the tool, sound mappings, and program comprehension studies. Advanced ideas may provide a synthesis of those three areas.

Proposition	Conclusion
1. A consistent, comprehensible mapping of non-speech sound patterns to the static entities of a software system can be devised.	A non-speech sound mapping from auditory signs and sequences of auditory signs to structural Java constructs, referred to as the reference mapping, was devised. That mapping is consistent in structural and acoustical ways. It is comprehensible when applied to relatively small programs, as demonstrated in the first study and verified in the second study.
2. The mapping can be used to identify software entities.	The ability to identify specific software entities is limited. This may hinder the application of the reference mapping or a similar sound mapping to larger programs. On the other hand, longer periods of training and usage than were possible over the course of the two studies may enhance identification of key entities through repetition, and may accordingly help overcome application to larger programs.
3. The mapping can be used to characterize software entities and their relationships when encountered.	Successful characterization of entities as packages, classes, interfaces, and methods was demonstrated in both studies. Moreover, successful characterization of classes according to language-syntactic analogues (e.g., static) and semantics (e.g., data writer, main method) is also successful. Listeners can make use of localization to determine at least simple characteristics such as whether the sound represents a local or external entity. In the task-based experiment, subjects were easily able to ascertain the relationship of one entity to others when they could recall the others' sounds. Otherwise, determining the relationship
4. The mapping, incorporated into an integrated software development environment, can be used in the performance of program comprehension tasks.	The task-based experiment has shown the reference mapping supported by an audiovisual tool to be usable. That is, tasks were completed using the sound mapping in place of an efficient visual search feature and in addition to other visual features of Eclipse. It is expected that this conclusion extends to similar languages and integrated development environments. Retention during task performance appears to be stronger when the audio representation is concrete rather than abstract.
5. Use of the mapping in a multimodal software development environment can improve performance of software comprehension tasks over that using a software development environment without sound.	This has not been convincingly demonstrated. Reduced task duration when the tasks and environment meet certain conditions is suggested by one task (CP 4) in the experiment, but there is insufficient data to conclude that as valid.

Table 9.1: Propositions and Conclusions

9.4.1 Tool

The tool can be refined in ways that may further support program comprehension. The prototype tool can be improved for further experimentation by implementing the following features:

1. Enable the developer to listen to those on-screen and off-screen entities referenced by the developer-selected entity.

2. Enable the developer to halt an aural sequence of entities.

3. Enable the developer to pause an aural sequence of entities.

4. Provide the ability to skip aural entity sequences, listening parent-child relationships in their entirety or only to the single entity of interest.

5. Extend sonification to source code in editor windows.

6. Trigger sound by pointing device hover rather than click, which will require addressing timing considerations (sounds should not start too early or too late while performing rollover quickly and pausing on various entities).

7. Support transitive entity references.

8. Supplement the tool with an automated means to instantiate the sound mapping in order to be of practical use over numerous programs. The current manual method is excessively labor intensive. There may be research issues encountered in formulating the decision-making process for such automation. Some of the issues to be addressed include:

    - a sound selection algorithm that maximizes differentiation while maintaining structure and meaning,
    - accommodating developer preferences and possibly developer sound selection within a prescribed framework,

- incorporating semantic knowledge, for example, design patterns, an object as a data writer,
- updating entities and their related entities automatically as source code is being modified.

9.4.2 Mapping Sound

The science of auditory display design is far from mature, and the practice of audio display design is largely intuitive. Theory is emergent, as demonstrated by recent literature which provides only the first large-scale attempt to derive common design patterns [49] and the first significant steps to marry auditory display design and cognitive psychoacoustics [112]. Advances in the science of auditory display can inform the evolution of sound mappings with the reference mapping as a starting point. Future work in the sound mapping domain is discussed below.

1. The rules and design choices should be periodically re-examined from cognitive and empirical perspectives and adjustments made.
2. A useful field study would involve persons in various software engineering roles using a refined tool over time on a larger project than those in the present research.
    - If appropriate to the situation, researchers should monitor tool usage through keystroke and sound activation logging.
    - A subset of the sound to entity associations should be selected or even designed by the tool users.
    - Interviews and observations can supplement the acquired data to address questions such as for what tasks the tool is most frequently used, whether there is any actual or perceived task improvement, and retention of associations after substantial use.

3. The field study may be followed up by a task-based experiment similar to the second study in the present research, the difference being that the latter subjects would be have existing experience with mapping and tool and may have adapted task strategies to accommodate aural input. In one or more such experiments, various task types and longer task lengths should be chosen, and maintenance coding may be required.

4. The mapping rules generalized from the reference mapping should be validated. This could be accomplished in the experiment or field study described above.

9.4.3 Advanced Ideas

1. As new technologies become less expensive and more widely available, program comprehension using auditory means can take advantage of them. Specifically, binaural sound localization technology based on head-relative tracking functions, head tracking devices, and digital signal processing hardware and software may provide an added dimension of value. It may help to make program comprehension a more immersive experience. Audio techniques may be combined with virtual reality software visualization, providing a fully immersive program comprehension environment. Such an environment may ultimately replace the interactive development environment experience currently known to software developers. Software objects would "live" in 3D space, both visually and aurally.

2. A sound mapping divorced from visual means may service visually-impaired software engineers. Aural navigation, filtering, and details on demand would have to supplement the sound mapping. Details via spoken text may supplement the non-speech sounds. The work of Metatla, who constructed relational diagrams for the visually impaired, Nickerson, who sonified the

London Underground Real-Time Disruption Map, and associated research should be consulted [107][115].

3. A sound mapping and a refined tool may be employed as an educational aid. Computer science undergraduate programs have historically concentrated on languages, data structures, and algorithms rather than architectural concerns. The tool may be used in undergraduate programs to enhance students' awareness of software organization at the architectural level without major curriculum changes. The students would be asked to add or modify an algorithm or other small set of code in the context of a larger program, using sound to help understand what entities to integrate to. Awareness of separation of internal versus external libraries such as *java.io* would also be enhanced.

4. The present research addresses usage of the tool, but not construction of the sound mapping for each program to be investigated. Construction should be automated, leaving no work to the software developer, who has the program itself to deal with. Construction should also transparently accommodate changes made to the program. Construction techniques are destined to involve heuristic as well as deterministic decision making.

5. A sound mapping similar to the reference mapping may be applicable in other domains. Sonifying non-software domains and performing task-based evaluation would inform usability, advantages, and generalizations.

9.5 Summary

This chapter has offered conclusions and traced them to the research question and propositions introduced in Chapter 1. Suggestions for future work were also presented.

The thesis as a whole introduced a new concept for use of sonification to aid in the comprehension of static program structure, described a tool that implements the concept, offered an exploratory human-subject study to informally test and advance that concept, and followed that by a human-subject, task-oriented experiment to test the efficacy of the concept in a particular situation. While the experiment showed no significant improvement over traditional, non-audio means of performing program comprehension, one of its components suggested possible improvement under certain circumstances. Because of that, and because of the exploratory stage of the research and young maturity of the idea, the possibility is alive that the concept or some derivative of it may improve or influence software engineering practice. It is the researcher's hope that the ideas articulated in this thesis will spawn further ideas as yet unknown and encourage their study.

Appendices

Appendix A

Study One Session Protocol

This appendix contains the protocol used to guide the subject sessions. The protocol is meant to offer guidance that can be varied in response to per-subject variation and conditions that may arise during the session.

Participant Protocol

The human-participant test sequence is given below. The person administering the test should be sure the participant understands their role and agrees in writing to the conditions of the test prior to commencement of the steps below.

A. Ensure that Participant Knows Sufficient Computing

Ask the participant to describe the following Java constructs or their equivalents in another object-oriented language:

1. Meaning of package, class, method, and interface (essential)
2. What a constructor is (essential)
3. Difference between a static method and an instance method (essential)
4. What the "this" keyword means (essential)
5. What an interface is (essential)

6. A passive class versus an active class (desirable)

B. Administer Tone Recognition Test

1. Announce each group of items as shown in the answer key.
2. Play each of the fourteen items, pausing after each item.
3. For each item, record the Participant's response before proceeding to next item.

C. Play Training Stream

1. Provide Simple Package Explorer view for inspection throughout this step.
2. Provide Mapping Guide for inspection throughout this step.
3. Play the training stream for the participant.
4. Allow the participant to ask any questions, pausing the stream.
5. When done, allow the participant to ask questions, replay part of the stream, or replay the entire stream.

D. Play Expenses stream

1. Begin recording the session.
2. Ask participant to identify each element's type and characteristics. Early in the stream, pause as needed. Later in the session, bias toward not pausing.
3. After hearing a package and all of its elements, pause and ask the participant to describe the structure of the package and the elements it contains.
4. At pauses, ask the structural significance or meaning of groups of elements.

E. F. G. Play the two References Streams and the Class Size Stream

1. For each stream, ask questions as above.
2. For the references streams, ask which referenced items are external vs. internal. Also ask any meaning, in an architectural sense, of what is called. (The external class and its methods are all about data writing.)
3. For the class size stream, ask the size range of each class heard.

H. Draw project structure diagram

1. Provide the participant with paper and pencil.
2. Re-playing the streams heard in E-G above, have the participant draw the structure of the project in a comfortable form: a tree, UML diagrams, or other. The participant may re-play parts of the stream as needed.

Interview

1. Elicit any further observations and impressions that the participant may have. Seed the interview with the following questions:
   (a) What was easy and what was hard for you to do?
   (b) Would you use a tool based on this sound mapping? Why or why not?
   (c) In what circumstances, or in support of what activity, would you envision using the tool?
   (d) What further thoughts strike you?

Appendix B

Study Two In-Session Training

B.1 Description

The protocol for in-session training is included below. Subjects are encouraged to ask questions during the training part of the session. A topic can be skipped if it has been adequately addressed prior to being encountered in the protocol.

B.2 Protocol

B.2.1 Set Up

Bring up and initialize the sonified Expenses project in Eclipse. Expand/contract stuff such that the three classes appear in the Explorer, but no classes or methods do. Ensure that diagram is present.

B.2.2 Introduction

- Explain the space-air-earth analog for packages, classes/interfaces, and methods.
- Have the subject open and close various things in the Eclipse project to get the feel for navigating among packages, classes, interfaces, and meth-

ods. Explain the space-air-earth analog for packages, classses/interfaces, and methods.

B.2.3 Packages

- Explain that a package has an underlying satellite-like sound. Play the underlying sound.
- Have the subject play the three package sounds using the myself selection.

B.2.4 Classes and Interfaces

- Have the subject expand client1Package and then Expenses.java.
- Have the subject play Expenses.java. Explain that the wind-like sound is the underlying class sound, and the single tone is the unique sound identifying the Expenses class.
- Have the subject play Expenses, the class listed immediately underneath Expenses.java. Ensure that the subject understands that they are both the same item.
- Have the subject expand commonPackage and expensesPackage (but not their classes).
- Have the subject play three or four of the newly-appearing classes. (If the subject stumbles upon an interface, explain it.)
- Have the subject play the method expensesPackage.ExpenseAccess.store. Explain that the purely upward pattern of a single instrument indicates a data writer: a method whose sole function is to take some data and write it. Explain the analogy of an upward pattern as posting something up. Mention that the converse is a data reader, with a purely downward, single-instrument pattern. This can be any instrument: a flute, an electronic

instrument, etc. Also note this is NOT an accessor/mutator method, which has its own bell-like sound (lengthier than the sound for the present store method.)

- Explain that interfaces are each unique bird calls. Have the student play INonTaxable.java (or the interface INonTaxable) and ITaxable.java (or the interface ITaxable).

- Have the subject play the class NonTaxable[.java]. Explain that an interface bird call can be placed after a class identifying sound, while still superimposed on the underlying wind sound, to indicate that the class implements that interface. Have the subject play Taxable[.java].

- Not found: Have the student play the TaxNotIncluded class. Explain that this is the sound not found sound. It applies equally to packages, interfaces, classes, and methods. The entity exists, but its sound cannot be determined.

- Have the subject play the class ExpenseList. Note that it implements an interface external to the project. That interface turns out to be java.util.Iterator.

B.2.5 Methods

- Have the subject play the constructors for Taxable and NonTaxable. Explain that constructors are outdoor wood hammering sounds.

- Have the subject play the two constructors for ExpenseDelimitedAccess. Explain that overloaded constructors have different numbers of (more) hammer strokes. Have the subject play the constructor for ExpenseList. Note that it is the same as that for ExpenseDelimitedAccess. Also point out that the items above ExpenseList() are variables and are therefore not sonified.

- Have the subject play Expenses:main(String[]). Explain that the anvil before the primary method sound indicates that the method is static.

- Accessors and Mutators. Have the subject play expensesPackage : AppConfiguration : getAccessMethod(). Ask if the subject recalls what the anvil means. (answer: static.) Explain that the double bell sound indicates an accessor or mutator. The lower of the two double bells, which also declines slightly in pitch, means accessor, while the higher one, which rises slightly in pitch, means mutator. Have the subject open getAccessMethod() in an editor window to verify that it is static and that it is an accessor. Have the subject play putAccessMethod() and view it in the editor to verify it is a mutator.

- Readers and Writers. Have the subject play expensesPackage : AppConfiguration : store(ExpenseData). Explain that the purely upward pattern indicates a data writer, whether to a database, disk, or other permanent storage. Explain that a reader is purely downward. Also explain that the upward or downward pattern must be a single sound, not an ensemble (such as an orchestra) or combined sounds. Have the subject play expensesPackage : ExpenseFacade : retrieveAll(). Explain that this is a data reader. Explain that it meets the criteria by being a single tone in a strictly downward pattern. Have the subject play retrieveByDay(Date). Ask what the sound means. (It means that the sound is unidentified.)

- Accessors/Mutators vs. Readers/Writers. Ensure that the subject can differentiate readers, writers, accessors, and mutators by playing them without the subject looking and making the subject guess which is being played.

B.2.6 Extends, Instantiated-By, Referenced-By

- Extends. Have the subject turn on Item, select Itaxable.java, and listen to it. Next have the subject select Taxable.java and listen to it. Have the student explain what is heard: that Taxable is a class that implements the interface

ITaxable. Have the student select TaxIncluded.java and listen to it. Now have the student change from Item to What class does the item extend? and listen to TaxIncluded. Have the subject explain that TaxIncluded extends (inherits from) the superclassTaxable, which in turn implements the interface ITaxable. Be sure the subject understands the sound realization rules of extends versus implements.

- Extends, continued. Ask the subject to listen to select several other classes at random and listen to what they extend. Ensure that the student knows that, if nothing is heard, the class does not inherit from a superclass. Have the student locate, by listening, another class within expensesPackage, other than TaxIncluded and TaxNotIncluded, that extends a superclass, and identify what it inherits from. (ExpenseDelimitedAccess and ExpenseXmlAccess each inherit from ExpenseAccess).

- Instantiated by. NOTE: instantiated-by is only implemented for classes within the packages client1Package and commonPackage. Have the student switch to What calls the item and select a constructor to hear what calls it. This clearly means that the class is instantiated by the constructor's caller. But what if there are multiple constructors or no constructor? Explain that that's why What instantiates this item? is also available. Have the student switch to what instantiates this item? and select and play at least three classes to hear what instantiates them.

- Called by. NOTE: called-by is only implemented for methods within the packages client1Package and commonPackage. Have the subject select and commonPackage : DebugHelper and play its own sound.

- Have the subject switch to what calls the item? and play commonPackage. Explain the serialization of what calls it.

B.2.7 Within vs. Outside Project Space

- Called by infrastructure. Have the subject select the method client1Package : Expenses : main and listen to what calls it. Explain that this sound represents, generically, infrastructure, meaning the operating system, its event loop code, and such. Explain that it is characterized as outside the project space primarily by being off to either side and secondarily by sounding further away. Items internal to the project sound centered and closer. Mention that entities such as the class java.io and its methods are also outside the project space and therefore also have the off-center, more distant sound.

B.2.8 Practice

Perform each practice task with and without sound. Ensure that the subject knows about the search features within Eclipse. The subject should strictly use the sound mapping for the tasks to answer the questions in 1 through 6. The subject should use sound and then conventional means for 7.

- Various simple tasks for reinforcement. (Does class x implement an interface? What calls method x.y? Does class y extend any superclass?) Repeat each for multiple items x.
- What calls DebugHelper : getCount? What calls DebugHelper : incIdent?
- What instantiates commonPackage : DebugHelper? Why?
- What calls DebugHelper : warn(String) ? Describe what you hear.
- What instantiates DebugHelper? [answer: nothing does.] Why is DebugHelper never instantiated? [answer: all of its methods are static.] Is it used at all?

- Are the debug warning messages indicating unimplemented classes implemented in a consistent manner? How best would you make them consistent and ensure their continued consistency?

- The client gets an expense item from the user, but the server stores it. Explain how the information placed in an instance of ExpenseData is handled by the server upon receipt from the client. [Maximum task duration: 5 minutes.] Use sound where applicable.

- Demonstrate how the previous task would be performed without using sound.

Appendix C

Study Two Session Protocol

C.1 Description

The protocol for the Study Two experiment session follows. This protocol is performed after completion of in-session training and an optional ten to fifteen minute break.

C.2 Researcher Instructions

C.2.1 Have on Hand

- Timepiece that counts minutes and seconds (e.g., watch, stopwatch)
- Pad of paper and pen or pencil, for experimenter
- Pad of paper and pen or pencil, for subject
- Experiment description (in case subject did not read theirs beforehand)
- Ethics form (in case subject did not complete theirs beforehand)
- Pre-session questionnaire (in case subject did not complete theirs beforehand)

- Training script, for experimenter
- Sound mapping reminder diagram
- Printout of Simple Project for use during training
- Experimenter's worksheet

C.2.2 Then Do

1. Collect ethics form from the subject, and address any concerns about the experiment.
2. Collect questionnaire from the subject.
3. Determine if the subject is a professional or a student. In general, a postgraduate student who has performed a year or more of non-trivial programming work as a research assistant is considered a professional. Add the subject's name to the experimenter's worksheet and add the next sequential subject number for professional or student.
4. Determine which of the two programs will use sonification, using the experimenter's worksheet.
5. Determine the order in which the programs will be given and which will use sonification, using the experimenter's worksheet.
6. Ensure that the subject completed the pre-session training regimen:
    - Played the training audio stream
    - Played and performed the training audio exercises
    - Completed the Ethics Form
    - Completed the Pre-Session Questionnaire

7. If the regimen was not completed, the subject should complete it before the session. The subject should take a break of at least five minutes between the self-training regimen and the next step, in-person training using the Expenses project.

8. Provide the reminder diagram, bring up the Expenses project, and administer the in-person training. Follow the relevant steps in the *Common Procedure for Both Programs* and *Sonified Procedure for Both Programs* below to bring up the Expenses project.

9. Administer the first project, following the relevant procedure.

10. Administer the second project, following the relevant procedure.

C.2.3 Common Procedure for Both Programs

1. Bring up Eclipse. (/prototype/eclipsedev.sh)

2. Bring up the desired project. (Run Run Configurations Select Configuration)

3. Ensure that the Package Explorer window appears at the left, and ensure that it is wide enough to show all the class names. If does not appear, make it visible. (Window Show View Project Explorer)

4. Ensure that the Package Explorer shows the classes but not any of the methods. (Contract classes if necessary using the arrow to the left of each.)

5. Ensure that no source code appears in any visible editor frame.

6. Ensure that the Sonify View is visible. (Window Show View Other Sonification Sonify View)

7. Provide pad (8.5 x 11 inches or larger) and pen or pencil.

8. Perform either the SONIFIED PROCEDURE or UNSONIFIED PROCEDURE.

Sonified Procedure for Both Programs

1. Ensure that the audio output is not muted.
2. Start CSound by running the appropriate script.
    - sonifExp.sh for Expenses
    - sonifCourse.sh for Course Predictor
    - sonifPict.sh for Pictionary
3. Initialize the project, then initialize the sound stream. (You will hear the sound stream startup sound.)
4. Administer the in-person training using the Expenses project or administer exploration using the Course Predictor or Pictionary project.
5. Administer the tasks, observing and listening to the talk-aloud protocol.
6. Administer the post-project debrief.

Unsonified Procedure for Both Programs

1. Do not provide audio. (If desired, mute the audio output to ensure this.)
2. Allow the subject to explore for the duration specified for the project. Subject may use any visual means available.
3. Administer the tasks and a short debrief after each task.
4. Administer the post-project debrief.

For Each Task

Explain the task, repeating if asked until clear. For the last task for each project, place the task description near the subject. Inform the subject to issue a finished remark when the task has been completed (when the subject is confident of the result or answer). The subject may also issue a stuck remark when sure he or she is stuck before the maximum task duration. Start the clock, allowing the subject to perform the task up to the stated maximum time. Stop the clock upon the subject's "finished" or "stuck" remark. Note the time taken and any other significant observations. Close any open editor window before starting the subsequent task. Ensure that no classes are expanded prior to the subsequent task.

C.2.4 CP - Course Predictor

INTRODUCE THE PROJECT

Show the Course Predictor GUI, the applet version of which is at
http://www.loyola.edu/computerscience/graduate/index.html.
Explain that it filters such that one can see courses offered in different categories for different semesters.

EXPLORATION

Ask if the subject will feel uncomfortable being observed during Exploration. If so, do not observe the exploration. Otherwise, observe parts of the exploration to gain an idea whether bottom-up, top-down, or opportunistic comprehension appears to be occurring. Tell the subject they can freely explore for 12 minutes. Allow exploration for 12 minutes. Do not count any time spent asking and addressing clarification.

Ask if the subject will feel uncomfortable if you make an audio recording of

the remainder of the session. Explain that only the debrief after each task will be used. If comfortable, start the audio recording.

Briefly debrief the subject after each task as to their strategy and what they learned about the code.

TASKS, IN SEQUENCE

1. [5 minutes] What package/class/method combination(s) instantiate the class URLButton? (Answer: the method cpred.init.)

2. [5 minutes] Identify all classes and methods which are callers of the method cpred.greenPanel. (Answer: the method cpred.init.)

3. [5 minutes] Does URLButton implement any interfaces? If so, how many? Are they internal or external to the project? (Answer: it implements ActionListener, which is external.)

4. [5 minutes] Is URLButton:actionPerformed called by any code internal to the project? Is it called by any code external to the project? (Answer: it is called by the "infrastructure.")

5. [15 minutes] (Perfective) (First, describe to the subject the salient points of the class java.util.logging.Logger.)

   Previous developers have implemented logging of desired messages. Multiple developers have each worked on their own logging code, so we know it can be streamlined. Currently, logging may or may not meet the following requirements:

   - Requirement 1: "All messages that are logged will be logged to both the console and the log file."
   - Requirement 2: "All logging shall occur via a single logging class within the project."

- Requirement 3: "All logging shall utilize the built-in Java class java.util.logging.Logger."

Determine if the project meets the requirements stated above. If they are met, how? If they are not met, why? How can logging be streamlined?

(Solution: CLog only logs to the console, and it doesn't use the Java Logger class, but multiple cpred methods call it. Log does use the Java Logger class, and one method calls it. History extends Log, albeit improperly, and it is instantiated, but it is never subsequently called for any logging. To streamline, remove History and its instantiation, remove CLog, and redirect CLog's call targets to Log.)

PROJECT DEBRIEF

Interview the subject about their overall strategies, comfort level with use of sound, and overall impressions. When done, stop the audio recording.

C.2.5 PICT - Pictionary Server Package

INTRODUCE THE PROJECT

Show or sketch the Pictionary client GUI. Explain that there are rounds of turns in which each player draws while the others guess. The game provides a random word from a list to the player whose turn it is. The player draws in the large graphics area, and the others guess the word in the smaller text area. Either a player guesses and is awarded points or the timer runs out and the next turn begins.

EXPLORATION

1. Ask if the subject will feel uncomfortable being observed during Exploration. If so, do not observe the exploration. If not, observe parts of the exploration

to gain an idea whether bottom-up, top-down, or opportunistic comprehension appears to be occurring. Tell the subject they can freely explore for a few minutes. If SONIFIED, they can use sound and expand anything in the browser. They cannot bring up files in an editor window. If they do by accident, they must immediately close the editor window. If UNSONIFIED, they can do anything that Eclipse will let them do except use sound. Tell the subject they can freely explore for 5 minutes. Allow exploration for 5 minutes. Do not count any time spent asking and addressing any usage or clarification questions.

2. Ask the subject if they will feel uncomfortable if you make an audio recording of the remainder of the session. Explain that only the debrief after each task will be used. If comfortable, start the audio recording. Note, the tasks below apply only to items in the PictionaryServer package. Briefly debrief the subject after each task as to their strategy and what they learned about the code.

TASKS, IN SEQUENCE

1. [5 minutes] Identify all callers of the following method:
   PictionaryServer.WordRepository.getNextWord.
   (Answer: the method GameServer.run.)

2. [5 minutes] What instantiates WordRepository?
   (Answer: the constructor in the GameServer class.)

3. [5 minutes] Identify all data writers within the Pictionary Server package.
   (Answer: there are none. One might guess loadFile, whose representation has an upward pattern but is not a single sound, and which does more than simple write data.)

4. [5 minutes] Identify any/all dead classes and methods. (i.e., those that are never called.) Write the answer on paper.

   (Answer: four of them: getCurrentWord, getList, getNumberOfWords, and getWordsLeft.)

5. [15 minutes] (Quality Assurance task)

   Ensure that the code meets the following two design criteria:

   "The entire word list will be made available in randomized order

   (a) before the first round is begun, and (b) after every 5 rounds of turns."

   If the code does not meet a criterion, explain why not. If the code meets a criterion, explain how it does.

   (Solution: The word list is only refreshed upon initialization, so only (a) is met. WordRepository.shuffleWords is called by WordRepository.loadFile, which is in turn called by the WordRepository constructor, which is only called by the GameServer constructor, which is only called by GamerServer-Main:main.)

PROJECT DEBRIEF

Debrief the subject about their overall strategies, comfort level with use of sound, and overall impressions. When done, stop the audio recording.

C.2.6 PSSUQ

Administer the PSSUQ after both sets of tasks have been performed.

Bibliography

[1] S. Adolph, W. Hall, and P. Kruchten, "Using grounded theory to study the experience of software development," *Empirical Software Engineering*, vol. OnlineFirst, 26 January 2011.

[2] R. Angel, "Pictionary," Game. Seattle Games, Inc., 1985.

[3] Audacity Project. Audacity web site. Last accessed 17 April 2011. [Online]. Available: http://audacity.sourceforge.net/

[4] R. Baecker, C. DiGiano, and A. Marcus, "Software visualization for debugging," *Communications of the ACM*, vol. 40, no. 4, pp. 44–54, 1997.

[5] M. Barra, "Personal Webmelody: Customized sonification of web servers," in *Proceedings of the 7th International Conference on Auditory Display (ICAD'01)*, Helsinki, Finland, July 29 - Aug 1 2001.

[6] S. Barrass, "Sonification design patterns," in *Proceedings of the* 9^{th} *International Conference on Auditory Display (ICAD'03)*, Boston, USA, July 6-9 2003, pp. 131–145.

[7] V. Basili, R. Selby, and D. Hutchens, "Experimentation in software engineering," *IEEE Transactions on Software Engineering*, vol. 12, no. 7, pp. 733–743, July 1986.

[8] D. Bergantz and J. Hassell, "Information relationships in PROLOG programs: how do programmers comprehend functionality?" *International Journal of Man-Machine Studies*, vol. 34, pp. 313–328, 1991.

[9] T. Berling and T. Thelin, "An industrial case study of the verification and validation activities," in *Proceedings of the* 9^{th} *International Symposium on Software Metrics*, Molndal, Sweden, Sept. 3-5 2003, pp. 226–238.

[10] L. Berman, S. Danicic, K. Gallagher, and N. Gold, "The sound of software: Using sonification to aid program comprehension," in *Proceedings of the* 14^{th} *International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, June 14-16 2006.

[11] L. Berman and K. Gallagher, "Listening to program slices," in *Proceedings of the* 12^{th} *International Conference on Auditory Display (ICAD'06)*, London, UK, June 20-23 2006.

[12] ——, "Integrating CSound with Eclipse: Listening to software under development," *CSound Journal*, Winter 2009, http://www.csounds.com/journal/.

[13] ——, "Using sound to understand software architecture," in *Proceedings of the* 27^{th} *ACM International Conference on Design of Communication (SIGDOC'09)*, Bloomington, USA, October 5-7 2009, pp. 127–134.

[14] T. Biggerstaff, B. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the International Conference on Software Engineering (ICSE'93)*, Baltimore, USA, May 17-21 1993, pp. 482–498.

[15] M. Blattner, D. Sumikawa, and R. Greenberg, "Earcons and icons: Their structure and common design principles," *Human-Computer Interaction*, vol. 4, pp. 11–44, 1989.

[16] D. Boardman, V. Khandelwal, G. Greene, and A. Mathus, "LISTEN: a tool to investigate the use of sound for the analysis of program behavior," in *International Computer Software and Applications Conference (COMPSAC'95)*, Dallas, USA, Aug 9-11 1995, pp. 184–189.

[17] S. Boccuzzo and H. Gall, "Software visualization with audio supported cognitive glyphs," in *Proceedings of the 24th International Conference on Software Maintenance*, Beijing, China, Sep. 28 - Oct. 4 2008.

[18] ——, "CocoViz with ambient audio software exploration," in *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 16-24 2009.

[19] D. Bock, "ADSL: An auditory domain specification language for program auralization," in *Proceedings of the* 2nd *International Conference on Auditory Display (ICAD'94)*, Sante Fe, USA, Nov. 7-9 1994, pp. 251–256.

[20] B. Boehm, "A spiral model of software development and enhancement," *ACM SIGSOFT Software Engineering Notes*, vol. 11, no. 4, pp. 14–24, August 1986.

[21] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Reading, USA: Addison Wesley, October 20 1998.

[22] J. Boodhoo. (2006) Model view presenter. Last accessed 2 April 2011. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc188690.aspx

[23] R. Boulanger. The Csound Catalog. CD ROM, 2007. [Online]. Available: http://www.csounds.com/shop/csound-catalog/

[24] ——. The official CSound website. Last accessed 17 April 2011. [Online]. Available: http://www.csounds.com/

[25] S. Brewster, P. Wright, and A. Edwards, "Experimentally derived guidelines for the creation of earcons," in *Adjunct Proceedings of HCI'95*, Huddersfield, UK, August 1995, pp. 155–159.

[26] F. Brooks, "No silver bullet: Essence and accidents of software engineering," *IEEE Computer*, vol. 20, no. 4, pp. 10–19, April 1987.

[27] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, pp. 543–554, 1983.

[28] L. Brown and S. Brewster, "Drawing by ear: Interpreting sonified line graphs," in *Proceedings of the* 9^{th} *International Conference on Auditory Display (ICAD'03)*, Boston, USA, July 6-9 2003, pp. 152–156.

[29] M. Brown, "Color and sound in algorithm animation, research report 76a," DEC Systems Research Center, Palo Alto, USA, Tech. Rep., 1991.

[30] ——, "An introduction to Zeus: Audiovisualization of some elementary sequential and parallel sorting algorithms," in *Proceedings of the SIGCHI Conference on Human factors in Computing Systems (CHI'92)*, Monterey, USA, May 3-7 1992, pp. 663–664.

[31] S. Card, T. Moran, and A. Newell, *The Psychology of Human-Computer Interaction.* Hillsdale, US: Lawrence Erlbaum Associates, 1983.

[32] N. Chapin, J. Hale, K. Khan, J. Ramil, and W. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, pp. 3–30, 2001.

[33] E. Childs and V. Pulkki, "Using multi-channel spatialization in sonification: a case study with meterological data," in *Proceedings of the* 9^{th} *International Conference on Auditory Display (ICAD'03)*, Boston, USA, July 6-9 2003, pp. 192–195.

[34] R. Clarke, "Appropriate research methods for electronic commerce," *Author's research web site, http://www.rogerclarke.com/EC/ResMeth.html*, 19 April 2000.

[35] G. Coleman and R. O'Connor, "Using grounded theory to understand software process improvement: A study of Irish software product companies," *Information and Software Technology*, vol. 49, no. 67, pp. 654–667, June 2007.

[36] D. Cooper and P. Schindler, *Business Research Methods,* 8^{th} *ed.* Boston, USA: McGraw-Hill Irwin, 2003.

[37] C. Corritore and S. Wiedenbeck, "Mental representations of expert procedural and object-oriented programmers in a software maintenance task," *International Journal of Human-Computer Studies*, vol. 50, no. 1, pp. 61–83, 1999.

[38] D. Coulter, *Digital Audio Processing.* Lawrence, KS, USA: CMP Books, 2000.

[39] Cycling '74. MAX/MSP web site. Last accessed 17 April 2011. [Online]. Available: http://www.cycling74.com/products/

[40] S. Das, W. Lutters, and C. Seaman, "Understanding documentation value in software maintenance," in *Proceedings of the 2007 Symposium on Computer Human Interaction for the Management of Information Technology (CHIMIT'07)*, Cambridge, USA, March 30-31 2007.

[41] T. Dingler, J. Lindsay, and B. Walker, "Learnability of sound cues for environmental features: auditory icons, earcons, spearcons, and speech," in *Proceedings of the* 14^{th} *International Conference on Auditory Display (ICAD'08)*, Paris, France, June 23-27 2008.

[42] DOD-STD-2167A, *Military Standard, Defense System Software Development.* USA: Department of Defense, 1988.

[43] C. Dodge and T. Jerse, *Computer Music: Synthesis, Composition, and Performance, 2nd Ed.* New York, USA: Schirmer, 1997.

[44] S. Easterbrook, J. Singer, M. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. Sjoberg, Eds. London, UK: Springer-Verlag, 2008, pp. 285–311.

[45] Eclipse Foundation. The Eclipse website. Last accessed 17 April 2011. [Online]. Available: http://www.eclipse.org/

[46] M. Ferati, S. Mannheimer, and D. Bolchini, "Acoustic interaction design through audemes: Experiences with the blind," in *Proceedings of the 27th ACM International Conference on Design of Communication (SIGDOC'09)*, Bloomington, USA, October 5-7 2009, pp. 23–28.

[47] J. Finlayson and C. Mellish, "The audioview - providing a glance at Java source code," in *Proceedings of the 11th International Conference on Auditory Display (ICAD'05)*, Limerick, Ireland, July 6-9 2005.

[48] J. Francioni, l. Albright, and J. Jackson, "Debugging parallel programs using sound," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, USA, May 20-21 1991, pp. 68–75.

[49] C. Frauenberger, "Auditory display design: An investigation of a design pattern approach," Ph.D. dissertation, Queen Mary Campus, Univeristy of London, London, UK, 2009.

[50] C. Frauenberger, V. Putz, and R. Höldrich, "Interaction patterns for auditory user interfaces," in *Proceedings of the 11th International Conference on Auditory Display (ICAD'05)*, Limerick, Ireland, July 6-9 2005.

[51] R. Gallupe, A. Dennis, W. Cooper, J. Valachich, L. Bastianutti, and J. Nunamaker, "Electronic brainstorming and group size," *Academy of Management Journal*, vol. 35, no. 2, pp. 350–360, 1992.

[52] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Boston, USA: Addison-Wesley, 1995.

[53] W. Gaver, "Auditory icons: Using sound in computer interfaces," *Human-Computer Interaction*, vol. 2:2, pp. 167–177, 1986.

[54] ——, "The SonicFinder: An interface that uses auditory icons," *Human-Computer Interaction*, vol. 4:2, pp. 67–94, 1989.

[55] M. Gilfix and A. Couch, "Peep (the network auralizer): Monitoring your network with sound," in *Proceedings of the USENIX System Administration Conference (LISA)*, New Orleans, USA, December 3-8 2000, pp. 109–117.

[56] N. Gold, "Hypothesis-based concept assignment to support software maintenance," Ph.D. dissertation, University of Durham, Durham, UK, 2000.

[57] N. Gold and K. Bennett, "Hypothesis-based concept assignment in software maintenance," *IEEE Proceedings - Software*, vol. 149, no. 4, pp. 103–110, August 2002.

[58] A. Goldberg, "Collaborative software engineering," *Journal of Object Technology*, vol. 1, no. 1, pp. 1–19, May 2002.

[59] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification, 3^{rd} Edition*. Boston, USA: Addison-Wesley, 2005.

[60] R. Graham, "Use of auditory icons as emergency warnings: evaluation within a vehicle collision avoidance application," *Ergonomics*, vol. 42, no. 9, pp. 1233–1248, 1999.

[61] D. Grout, *A History of Western Music, Revised Edition.* New York: Norton, 1973.

[62] F. Halsall, *Data Communications, Computer Networks, and Open Systems, 3^{rd} Edition.* Workingham, UK: Addison-Wesley, 1992.

[63] C. Harding, I. Kakadiaris, J. Casey, and R. Loftin, "A multi-sensory system for the investigation of geoscientific data," *Computers & Graphics*, vol. 26, no. 2, pp. 259–269, April 2002.

[64] F. Hayes and D. Coleman, "Coherent models for object-oriented analysis," in *Conference proceedings on Object-oriented programming systems, languages, and applications (OPPSLA'91)*, Phoenix, USA, October 6-11 1991.

[65] A. Highsmith, "What is agile software development?" *CrossTalk, the Journal of Defense Software Engineering*, vol. 28, no. 8, pp. 4–9, October 2002.

[66] S. Hove and B. Anda, "Experiences from conducting semi-structured interviews in empirical software engineering research," in *Proceedings of the 11^{th} International Software Metrics Symposium (METRICS 2005)*, Como, Italy, September 19-22 2005, pp. 23–32.

[67] K. Hussein, E. Tilevich, S. Kim, and I. Bukvic, "Sonification design guidelines to enhance program comprehension," in *17^{th} IEEE International Conference on Program Comprehension (ICPC'09)*, Vancouver, Canada, May 17-19 2009, pp. 120–129.

[68] IBM. Rational Rose. Last accessed 7 August 2011. [Online]. Available: http://www-01.ibm.com/software/awdtools/developer/rose/

[69] International Community for Auditory Display, "ICAD web site," last accessed 16 April 2011. [Online]. Available: http://www.icad.org/

[70] ISO Std 12207, *Systems and software engineering - software life cycle processes, 2nd ed.* www.iso.org: International Standards Organization, 2008.

[71] ISO Std 14764, *Information technology - Software maintenance.* www.iso.org: International Standards Organization, 2006.

[72] D. Jameson, "Building real-time music tools visually with Sonnet," in *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, Boston, USA, June 10-12 1996, pp. 253–265.

[73] J. Jeans, *Science & Music.* New York: Dover, 1968, originally published 1937.

[74] jGrasp Project. jGrasp web site. Last accessed 02 January 2011. [Online]. Available: http://www.jgrasp.org/

[75] L. Karlsson, A. Dahlstedt, B. Regnell, J. Natt, and A. Persson, "Requirements engineering challenges in market-driven software development - an interview study with practitioners," *Information and Software Tehnology*, vol. 49, pp. 588–604, February 2007.

[76] J. Keller, "Educational testing of an auditory display regarding seasonal variation of martial polar ice caps," in *Proceedings of the 9th International Conference on Auditory Display (ICAD'03)*, Boston, USA, July 6-9 2003, pp. 212–215.

[77] T. Kelly and J. Buckley, "A context-aware analysis scheme for Bloom's taxonomy," in *Proceedings of the 14th International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, June 14-16 2006, pp. 275–284.

[78] M. Kennedy, *The Concise Oxford Dictionary of Music, 3rd ed.* London, UK: Oxford University Press, 1980.

[79] D. Kieras, "Using the keystroke-level model to estimate execution times," *Department of Psychology, University of Michigan*, 2001, unpublished report.

[80] J. Kildal and S. Brewster, "Explore the matrix: Browsing numerical data tables using sound," in *Proceedings of the* 11^{th} *International Conference on Auditory Display (ICAD'05)*, Limerick, Ireland, July 6-9 2005.

[81] B. Kitchenham, "Evaluating software methods and tools part 1: The evaluation context and evaluation methods," *Software Engineering Notes*, vol. 21, no. 1, pp. 11–15, January 1996.

[82] B. Kitchenham and S. Pfleeger, "Personal opinion surveys," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. Sjoberg, Eds. London, UK: Springer-Verlag, 2008, pp. 63–92.

[83] C. Knight, "Virtual but visible software," in *Proceedings of the IEEE International Conference on Information Visualisation*, July 19-21 2000, pp. 198–205.

[84] ——, "Virtual software in reality," Ph.D. dissertation, University of Durham, Durham, UK, 2000.

[85] J. Koenemann and S. Robertson, "Expert problem solving strategies for program comprehension," in *Proceedings of SIGCHI Conference on Human Factors in Computing Systems*, April 1991, pp. 125–130.

[86] J. Kontio, J. Bragge, and L. Lehtola, "The focus group as an empirical tool in software engineering," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. Sjoberg, Eds. London, UK: Springer-Verlag, 2008, pp. 93–116.

[87] G. Kramer and B. W. et. al., *Sonification Report: Status of the Field and Research Agenda.* Report prepared for National Sci-

ence Foundation: International Community for Auditory Display, 1999, http://www.icad.org/websiteV2.0/References/nsf.html.

[88] M. Lehman and J. Fernandez-Ramil, "Software evolution," *Software Evolution and Feedback: Theory and Practice, N. Madhavjii, J. Fernandez-Ramil, and Dewayne Perry, ed.*, pp. 7–40, 2006.

[89] M. Lehman and F. Parr, "Program evolution and its impact on software engineering," in *Proceedings of the* 2^{nd} *International Conference on Software Engineering*, San Francisco, USA, October 13-15 1976, pp. 350–357.

[90] G. Leplâtre and S. Brewster, "Designing non-speech sounds to support navigation in mobile phone menus," in *Proceedings of the* 6^{th} *International Conference on Auditory Display (ICAD'00)*, Atlanta, USA, April 2-5 2000, pp. 190–199.

[91] T. Lethbridge, "What knowledge is important to a software professional?" *IEEE Computer*, vol. 33, no. 5, pp. 44–50, 2000.

[92] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and Software*, vol. 7, pp. 325–339, 1987.

[93] J. Lewis, *IBM Computer Usability Satisfaction Questionnaires: Psychometric Evaluation and Instructions for Use, Technical Report 54.786.* Boca Raton, USA: IBM Human Factors Group, 1993.

[94] B. Lientz and E. Swanson, *Software Maintenance Management.* Reading, USA: Addison-Wesley, 1980.

[95] B. Lientz, E. Swanson, and G. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, June 1978.

[96] D. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," *Empirical Studies of Programmers*, vol. 1, pp. 80–98, 1986.

[97] P. Lucas, "An evaluation of the communicative ability of auditory icons and earcons," in *Proceedings of the* 2^{nd} *International Conference on Auditory Display (ICAD'94)*, Sante Fe, USA, June 20-24 1994, pp. 121–128.

[98] A. Martelli, *Python in a Nutshell, second edition*. Sebastopol, CA, USA: O'Reilly, 2006.

[99] MathSoft, *S-Plus4 Guide to Statistics*. Seattle, USA: MathSoft, Inc., 1997.

[100] ——, *S-Plus4 Programmer's Guide, Version 4.0*. Seattle, USA: MathSoft, Inc., 1997.

[101] A. Mathur. Project Listen/JListen web site. Last accessed 17 April 2011. [Online]. Available: http://www.cs.purdue.edu/homes/apm/listen.html

[102] M. Matthews, *The Technology of Computer Music*. Cambridge, USA: M.I.T. Press, 1969.

[103] B. Maxwell and H. Delaney, *Designing Experiments and Analyzing Data*. Mahwah, USA: Lawrence Erlbaum Associates, 2004.

[104] J. McAvoy and T. Butler, "The impact of the Abilene Paradox on double-loop learning in an agile team," *Information and Software Technology*, vol. 49, no. 6, pp. 552–563, June 2007.

[105] J. McCartney. SuperCollider author's site. Last accessed 29 January 2011. [Online]. Available: http://www.audiosynth.com/

[106] G. McCluskey. (1998) Using Java Reflection. Last accessed 29 March 2011. [Online]. Available: http://java.sun.com/developer/technicalArticles/ALT/Reflection/

[107] O. Metatla, N. Bryan-Kinns, and T. Stockman, "Constructing relational diagrams in audio: The multiple perspective hierarchical approach," in *Proceedings of the* 10^{th} *International ACM Conference on Computers and Accessibility (SIGACCESS)*, Halifax, Canada, Oct. 13-15 2008, pp. 97–104.

[108] Microsoft Corp. C# reference. Last accessed 29 March 2011. [Online]. Available: http://msdn.microsoft.com/en-us/library/618ayhy6.aspx

[109] MIDI Manufacturers' Association. (2001) Complete MIDI 1.0 detailed specification. Last accessed 17 April 2011. [Online]. Available: http://www.midi.org/techspecs/midispec.php

[110] G. Miller, "The magical number seven, plus or minus two," *Psychological Review*, vol. 63, pp. 81–97, 1956.

[111] H. Müller and K. Klashinsky, "Rigi - a system for programming in the large," in *Proceedings if the* 10^{th} *International Conference on Software Engineering (ICSE'88)*, Singapore, 1988, pp. 80–86.

[112] M. Mustonen, "A review-based conceptual analysis of auditory signs and their design," in *Proceedings of the* 14^{th} *International Conference on Auditory Display (ICAD'08)*, Paris, France, June 24-27 2008.

[113] NCSS, LLC. (2008) Power analysis and sample size (PASS). Kaysville, Utah, USA. URL last accessed 17 April 2011. [Online]. Available: http://www.ncss.com/pass.html

[114] M. Nees and B. Walker, "Model of auditory graph comprehension," in *Proceedings of the* 13^{th} *International Conference on Auditory Display (ICAD'07)*, Montreal, Canada, June 26-29 2007, pp. 266–273.

[115] L. Nickerson, T. Stockman, and J. Thiebaut, "Sonifying the London underground real-time disruption map," in *Proceedings of the* 13^{th} *International*

Conference on Auditory Display (ICAD'07), Montreal, Canada, June 26-29 2007.

[116] D. Norman, *The Design of Everyday Things, 2002 Edition.* New York, USA: Basic Paperback, 2002.

[117] Z. Obrenovic and D. Starcevic, "Modeling multimodal human-computer interaction," *IEEE Compuer*, vol. 37, no. 9, pp. 65–72, 2004.

[118] Open Sound Control Project. Open Sound Control web site. Last accessed 29 January 2011. [Online]. Available: http://opensoundcontrol.org/

[119] A. Osborn, *Applied Imagination, Principles and Procedures of Creative Thinking.* Charles Scribner's Sons, 1953.

[120] M. Patton, *Qualitative Research and Evaluation Methods,* 3^{rd} *Edition.* Thousand Oaks, USA: Sage Publications, 2002.

[121] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology*, vol. 19, no. 3, pp. 295–341, July 1987.

[122] S. Peres and D. Lane, "Sonification of statistical graphs," in *Proceedings of the* 9^{th} *International Conference on Auditory display (ICAD'03)*, Boston, USA, July 6-9 2003, pp. 157–160.

[123] S. Pfleeger, "Experimental design and analysis in software engineering, part 2: How to set up and experiment," *Software Engineering Notes*, vol. 20, no. 1, pp. 22–27, January 1995.

[124] ——, "Understanding and improving technology transfer in software engineering," *Journal of Systems and Software*, vol. 47, pp. 111–124, 1999.

[125] D. Phillips, "An introduction to OSC," *Linux Journal*, no. 175, Nov. 12 2008. [Online]. Available: http://www.linuxjournal.com/content/introduction-osc

[126] S. Pinker, *The Stuff of Thought*. London, UK: Penguin Books, 2007.

[127] A. Polli, "Atmospherics/Weather Works: A multi-channel storm sonification project," in *Proceedings of the* 10^{th} *International Conference on Auditory Display (ICAD'04)*, Sydney, Australia, July 6-9 2004.

[128] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings of the* 10^{th} *International Workshop on Program Comprehension (IWPC'02)*, Paris, France, June 27-29 2002, pp. 271–278.

[129] S. Redwine and W. Riddle, "Software technology maturation," in *Proceedings of the* 8^{th} *International Conference on Software Engineering (ICSE'85)*, 1985, pp. 189–200.

[130] F. Reed, "SBIR phase I final report: Data sonification project," CHI Systems, Inc., Lower Gwynedd, PA, USA, Tech. Rep., 2001, contract No. DAAB07-01-K604, US Army CECOM, Fort Monmouth, N.J.

[131] C. Roads, *Microsound*. Cambridge, USA: MIT Press, 2002.

[132] E. Rogers, *Diffusion of Innovations, Fifth Edition*. New York, USA: Free Press, 2003.

[133] C. Sadler and B. Kitchenham, "Evaluating software engineering methods and tools, part 4: The influence of human factors," *Software Engineering Notes*, vol. 21, no. 6, pp. 11–13, September 1996.

[134] M. Sarkar and M. Brown, "Graphical fisheye views," *Communications of the ACM*, vol. 37, no. 12, pp. 73–84, July 1994.

[135] F. Sauer. Eclipse metrics plugin 1.3.6. Last accessed 2 January 2011. [Online]. Available: http://sourceforge.net/projects/metrics

[136] J. Sayyad-Shirabad, T. Lethbridge, and S. Lyon, "A little knowledge can go a long way towards program understanding," in *Proceedings of the* 5^{th}

International Workshop on Program Comprehension (IWPC'97), Dearborn, USA, Mar. 28-30 1997, pp. 111–117.

[137] C. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, July 1999.

[138] ——, "Qualitative methods," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. Sjoberg, Eds. London, UK: Springer-Verlag, 2008, pp. 35–62.

[139] C. Seaman and V. Basili, "Communication and organization: an empirical study of discussion in inspection meetings," *IEEE Transactions on Software Engineering*, vol. 24, no. 7, pp. 559–572, 1998.

[140] M. Sensalire and P. Ogao, "How software visualization tools measure up," in *Proceedings of the* 5^{th} *International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa (AFRIGRAPH'07)*, Eastern Cape, South Africa, October 29-31 2007.

[141] T. Shaft, "Helping programmers understand computer programs: The use of metacognition," *Data Base Advances*, vol. 26, no. 4, pp. 25–46, November 1995.

[142] B. Shneiderman, "Dynamic queries for visual information seeking," in *Readings in Information Visaulization: Using Vision to Think*, S. Card, J. Mackinlay, and B. Shneiderman, Eds. San Francisco, USA: Morgan Kaufman, 1999, pp. 236–243.

[143] B. Shneiderman and C. Plaisant, *Designing the User Interface, Fourth Edition*. Boston, USA: Pearson Education, Inc., 2005.

[144] J. Singer and T. Lethbridge, "Studying work practices to assist tool design in software engineering," in *Proceedings of the* 6^{th} *International Workshop*

on Program Comprehension (IWPC'98), Ischia, Italy, June 24-26 1998, pp. 173–179.

[145] J. Singer, S. Sim, and T. Lethbridge, "Software engineering data collection for field studies," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. Sjoberg, Eds. London, UK: Springer-Verlag, 2008, pp. 9–34.

[146] M. Smith, "Runtime visualisation of object-oriented software," Ph.D. dissertation, University of Durham, Durham, UK, 2003.

[147] J. Sodnik, C. Dicke, S. Tomazic, and M. Billinghurst, "A user study of auditory versus visual interfaces for use while driving," *International Journal of Human-Computer Studies*, vol. 66, no. 5, pp. 318–332, May 2008.

[148] Software Engineering Institute, *Software Process Assessment Team Training (2 volumes)*. Pittsburgh, USA: Software Engineering Institute, Carnegie Mellon University, 1991.

[149] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on Software Engineering*, vol. 10, no. 5, pp. 595–609, September 1984.

[150] D. Sonnenwald, B. Gopinath, G. Haberman, W. Keese, and J. Myers, "InfoSound: an audio aid to program comprehension," in *Proceedings of the* 23^{rd} *Annaul Hawaii International Conference on System Sciences*, vol. 2, 1990, pp. 541–546.

[151] Sourceforge web site. Last accessed 20 January 2011. [Online]. Available: http://www.sourceforge.net/

[152] A. Stefik, K. Fitz, and R. Alexander, "Layered program auralization: Using music to increase runtime program comprehension and debugging effec-

tiveness," in *Proceedings of the* 14^{th} *International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, June 14-16 2006, pp. 89–93.

[153] A. Stefik and E. Gellenbeck, "Using spoken text to aid debugging: An empirical study," in *Proceedings of the* 17^{th} *International Conference on Program Comprehension (ICPC'09)*, Vancouver, Canada, May 17-19 2009, pp. 110–119.

[154] K. Stephan, S. Smith, R. Martin, S. Parker, and K. McAnally, "Auditory icons and deontative referents: implications for the design of auditory warnings," *Human Factors*, vol. 48, no. 2, pp. 288–299, Summer 2006.

[155] M. Storey, "A cognitive framework for describing and evaluating software exploration tools," Ph.D. dissertation, Simon Fraser University, Burnaby, Canada, 1998.

[156] M. Storey, L. Cheng, J. Singer, M. Muller, D. Myers, and J. Ryall, "How programmers can turn comments into waypoints for code navigation," in *Proceedings of the* 23^{rd} *International Conference on Software Maintenance (ICSM'07)*, Paris, France, October 2-5, 2007, pp. 265–274.

[157] M. Storey, F. Fracchia, and H. Muller, "Cognitive design elements to support the construction of a mental model during software visualization," in *Proceedings of the* 5^{th} *International Workshop on Program Comprehension*, Dearborn, USA, May 1997, pp. 17–28.

[158] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory,* 2^{nd} *Edition.* Thousand Oaks, USA: SAGE Publications, 1998.

[159] T. Strong, *Amendment No. 6600-2010-2 to FSM 6600 - Forest Service Manual - Systems Management.* Washington, USA: U.S. Department of Agriculture, May 11 2010.

[160] B. Stroustrup, *The C++ Programming Language, Third Edition.* Indianapolis, USA: Pearson, 2000.

[161] SuperCollider Project. SuperCollider web site. Last accessed 29 January 2011. [Online]. Available: http://supercollider.sourceforge.net/

[162] E. Swanson, "The dimensions of maintenance," in *Proceedings of the* 2^{nd} *International Conference on Software Engineering (ICSE'76)*, San Francicso, USA, 1976, pp. 492–497.

[163] S. Taft and R. Duff, *Ada95 Reference Manual.* Berlin, Germany: Springer-Verlag, 1997.

[164] C. P. Team, "Cmmi for development, version 1.3 (cmu/sei-2010-tr-033)," Software Engineering Institute, Pittsburgh, USA, Tech. Rep., November 2010.

[165] The Apache Ant Project. Ant web site. Last accessed 17 April 2011. [Online]. Available: http://ant.apache.org/

[166] The R Project. R project web site. Last accessed 19 March 2011. [Online]. Available: http://www.r-project.org/

[167] K. Turri, M. Mustonen, and A. Pirhonen, "Same sound - different meanings: a novel scheme for modes of listening," in *Proceedings of the* 2^{nd} *Audio Mostly Conference on Interaction with Sound*, Ilmenau, Germany, September 27-28 2007, pp. 13–18.

[168] M. Vargas and S. Anderson, "Combining speech and earcons to assist menu navigation," in *Proceedings of the* 9^{th} *International Conference on Auditory Display (ICAD'03)*, Boston, USA, July 6-9 2003, pp. 38–41.

[169] P. Vickers, "Ars informatica - ars electronica, improving sonification aesthetics," in *HCI 2005 Workshop on Understanding and Designing for Aesthetic Experience*, Edinburgh, UK, September 2005.

[170] P. Vickers and J. Alty, "When bugs sing," *Interacting with Computers*, vol. 14, pp. 793–819, 2002.

[171] ——, "Siren songs and swan songs: Debugging with music," *Communications of the ACM*, vol. 46, no. 7, pp. 86–93, July 2003.

[172] B. Vinz and L. Etzkorn, "A synergistic approach to program comprehension," in *Proceedings of the* 14^{th} *International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, June 14-16 2006, pp. 69–73.

[173] A. von Mayrhauser and S. Lang, "A coding scheme to support systematic analysis of software comprehension," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 526–540, 1999.

[174] A. von Mayrhauser and A. Vans, "From code understanding needs to reverse engineering tool capabilities," *Proceedings of the* 6^th *International Workshop on Computer-Aided Software Engineering (CASE'93)*, p. 230, July 19-23 1993.

[175] ——, "Program comprehension during software maintenance and evolution," *IEEE Computer*, vol. 28, no. 8, pp. 44–55, August 1995.

[176] ——, "Identification of dynamic comprehension processes during large scale maintenance," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 424–437, June 1996.

[177] A. von Mayrhauser, A. Vans, and A. Howe, "Program understanding behaviour during enhancement of large-scale software," *Software Maintenance: Research and Practice*, vol. 9, no. 5, pp. 299–327, 1997.

[178] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 332–357, 1984.

[179] S. Wiedenbeck, "The initial stage of program comprehension," *International Journal of Man-Machine studies*, vol. 35, pp. 517–540, 1991.

[180] C. Wilson and S. Lodha, "Listen: A data sonification toolkit," in *Proceedings of the* 3^{rd} *International Conference on Auditory Display (ICAD'96)*, Palo Alto, USA, Nov. 4-6 1996.

[181] U. Wiss and D. Carr, "An empirical study of task support in 3d information visualizations," in *Proceedings of the 1999 IEEE Conference on Information Visualization*, London, UK, July 1999, pp. 392–399.

[182] J. Wu, T. Graham, and P. Smith, "A study of collaboration in software design," in *Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE'03)*, Sept. 30 - Oct. 1 2003, pp. 304–313.

[183] R. Yin, *Case Study Research: Design and Methods,* 4^{th} *ed.* Thousand Oaks, USA: SAGE Publications, 2003.

[184] C. Zannier, M. Chiasson, and F. Maurer, "A model of design decision making based on empirical results of interviews with software designers," *Information and Software Technology*, vol. 49, no. 6, pp. 637–653, June 2007.

[185] H. Zhao, C. Plaisant, B. Shneiderman, and R. Duraiswami, "Sonification of geo-referenced data for auditory information seeking: Design principle and pilot study," in *Proceedings of the* 10^{th} *International Conference on Auditory Display (ICAD'04)*, Sydney, Australia, July 6-9 2004.

[186] I. Zmoelnig. The PureData Portal. Last accessed 17 April 2011. [Online]. Available: http://puredata.info/

[187] E. Zwicker and H. Fastl, *Psychoacoustics: Facts and Models*. Berlin, Germany: Springer-Verlag, 1990.